

# Handbook of Logic in Computer Science

UNIVERSITY OF ARIZONA



39001034133275

VOLUME 4

## Semantic Modelling

---

Edited by

S. ABRAMSKY, DOV M. GABBAY

and T. S. E. MAIBAUM



OXFORD SCIENCE PUBLICATIONS









Digitized by the Internet Archive  
in 2025







# HANDBOOK OF LOGIC IN COMPUTER SCIENCE

---

Editors

S. Abramsky, Dov M. Gabbay, and T. S. E. Maibaum



HANDBOOKS OF LOGIC IN COMPUTER SCIENCE  
*and*  
ARTIFICIAL INTELLIGENCE AND LOGIC  
PROGRAMMING

*Executive Editor*

Dov M. Gabbay

*Administrator*

Jane Spurr

---

Handbook of Logic in Computer Science

- Volume 1**    Background: Mathematical structures
- Volume 2**    Background: Computational structures
- Volume 3**    Semantic structures
- Volume 4**    Semantic modelling
- Volume 5**    Theoretical methods in specification and verification
- Volume 6**    Logical methods in computer science

Handbook of Logic in Artificial Intelligence and  
Logic Programming

- Volume 1**    Logical foundations
- Volume 2**    Deduction methodologies
- Volume 3**    Nonmonotonic reasoning and uncertain reasoning
- Volume 4**    Epistemic and temporal reasoning
- Volume 5**    Logic programming



# Handbook of Logic in Computer Science

Volume 4

Semantic Modelling

Edited by

S. ABRAMSKY

*Professor of Computing Science*

DOV M. GABBAY

*Professor of Computing Science*

and

T. S. E. MAIBAUM

*Professor of Foundations of  
Software Engineering*

*Imperial College of Science, Technology and Medicine  
London*

Volume Co-ordinator

S. ABRAMSKY

CLARENDON PRESS • OXFORD

1995

*Oxford University Press, Walton Street, Oxford OX2 6DP*

*Oxford New York*

*Athens Auckland Bangkok Bombay*

*Calcutta Cape Town Dar es Salaam Delhi*

*Florence Hong Kong Istanbul Karachi*

*Kuala Lumpur Madras Madrid Melbourne*

*Mexico City Nairobi Paris Singapore*

*Taipei Tokyo Toronto*

*and associated companies in*

*Berlin Ibadan*

*Oxford is a trade mark of Oxford University Press*

*Published in the United States by  
Oxford University Press Inc., New York*

*© The contributors listed on p. xv, 1995*

*All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, without the prior permission in writing of Oxford University Press. Within the UK, exceptions are allowed in respect of any fair dealing for the purpose of research or private study, or criticism or review, as permitted under the Copyright, Designs and Patents Act, 1988, or in the case of reprographic reproduction in accordance with the terms of licences issued by the Copyright Licensing Agency. Enquiries concerning reproduction outside those terms and in other countries should be sent to the Rights Department, Oxford University Press, at the address above.*

*This book is sold subject to the condition that it shall not, by way of trade or otherwise, be lent, re-sold, hired out, or otherwise circulated without the publisher's prior consent in any form of binding or cover other than that in which it is published and without a similar condition including this condition being imposed on the subsequent purchaser.*

*A catalogue record for this book is available from the British Library*

*Library of Congress Cataloging in Publication Data  
(Data available)*

*ISBN 0 19 853780 8*

*Typeset using LaTeX by Jane Spurr*

*Printed in Great Britain by  
Biddles Ltd,  
Guildford and King's Lynn*

# Preface

We are happy to present Volume 4 of the Handbook of Logic in Computer Science, on *Semantic Modelling*. The first two volumes of the Handbook presented the background on fundamental mathematical structures—consequence relations, model theory, recursion theory, category theory, universal algebra, topology; and on computational structures—term-rewriting systems,  $\lambda$ -calculi, modal and temporal logics and algorithmic proof systems. The computational structures considered thus far have been predominantly syntactic in character; while the discussion of mathematical structures has been quite general and free-standing.

In the present Volume and its companion Volume 3, these threads are drawn together. We look at how mathematical structures are used to model computational processes. In Volume 3, the focus is on general approaches: Domain theory, denotational and algebraic semantics, and the semantics of types. In the present Volume, some more specific topics are considered, and the emphasis shifts from structures to the actual modelling of some key computational features.

The first two Chapters treat concurrency. The first Chapter surveys the wide range of models for concurrent computation which have been proposed over the past 30 years. Precise comparisons and connections are made, using tools from Category theory. There is considerable emphasis on so-called “true concurrency” models, which take the independence of concurrent events as a fundamental feature of concurrent systems to be modelled directly, rather than by a reduction to non-deterministically interleaved sequential computation.

The second Chapter concerns the algebraic approach to concurrency theory which has come into increasing prominence over the past decade. This approach starts from the axiomatic standpoint, attempting to capture a variety of notions in concurrency by equational axioms over some algebraic signature. Methods from Rewriting theory, Model theory and Universal Algebra are then used to investigate the structural properties of these axioms.

Chapter 3 concerns the relationship between the denotational approach to semantics, which was the topic of a Chapter in Volume 3, and operational semantics, which gives an account of the meaning of a program directly in terms of its computational behaviour. Thus denotational semantics is more abstract, and usually exhibits a higher degree of mathematical structure,



whereas operational semantics gives a more implementation-oriented view. The correspondence between these two accounts of program meaning is of prime importance in the semantics of programming languages, as the bridge across which mathematical and computational insights can be transported. The Chapter takes the PCF language as a paradigmatic example. This language has been a *locus classicus* for the study of this topic over the past 20 years, and in particular of the key issue of Full Abstraction.

Chapter 4 studies effectiveness issues in semantics in an algebraic context. It builds on previous Chapters on Universal Algebra, Recursion theory, Topology and Domain theory.

Finally, Chapter 5 considers Abstract Interpretation, an important area of application of semantics, where theory meets practice. This concerns “Static Analysis” of programs: that is, determining properties of a program over some class of inputs without actually executing the program over all these inputs (which would in general be impossible or at least infeasible). This determination of properties is of prime importance in a variety of applications, including compiler optimization, program transformation, verification and parallelization. One main approach to static analysis, which both systematizes much *ad hoc* work on “program flow analysis”, and opens the way to some more sophisticated applications, is Abstract Interpretation. In this approach, one actually computes a simplified version of the denotational meaning of a program, and uses this simplified denotation to yield information about the property of interest. This work makes rather direct connections between theoretical work on programming language semantics, and some concrete applications in Software Engineering.

## The Handbooks

We see the creation of this Handbook and its companion, the *Handbook of Logic in Artificial Intelligence and Logic Programming* as a combination of authoritative exposition, comprehensive survey, and fundamental research exploring the underlying unifying themes in the various areas. The intended audience is graduate students and researchers in the areas of computing and logic, as well as other people interested in the subject. We assume as background some mathematical sophistication. Much of the material will also be of interest to logicians and mathematicians.

The tables of contents of the volumes were finalized after extensive discussions between Handbook authors and second readers. The first two volumes present the Background—Mathematical Structures and Computational Structures.

The chapters, which in many cases are of monographic length and scope, are written with emphasis on possible unifying themes. The chapters have an overview, introduction, and main body. A final part is dedicated to

more specialized topics.

Chapters are written by internationally renowned researchers in their respective areas. The chapters are co-ordinated and their contents were discussed in joint meetings. Each chapter has been written using the following procedures:

1. A very detailed table of contents was discussed and co-ordinated at several meetings between authors and editors of related chapters. The discussion was in the form of a series of lectures by the authors. Once an agreement was reached on the detailed table of contents, the authors wrote a draft and sent it to the editors and to other related authors. For each chapter there is a second reader (the first reader is the author) whose job it has been to scrutinize the chapter together with the editors. The second reader's role is very important and has required effort and serious involvement with the authors.

Second readers for this volume are:

Chapter 1: Concurrency—R. van Glabeek and P. S. Thiagarajan

Chapter 2: Process Algebra—J. A. Bergstra and J. W. Klop

Chapter 3: Correspondence—P.-L. Curien

Chapter 4: Effective Structures—J. A. Bergstra

Chapter 5: Abstract Interpretation—A. Mycroft and S. Hunt

2. Once this process was completed (i.e. drafts seen and read by a large enough group of authors), there were other meetings on several chapters in which authors lectured on their chapters and faced the criticism of the editors and audience. The final drafts were prepared after these meetings.
3. We attached great importance to group effort and co-ordination in the writing of chapters. The first two parts of each chapter, namely the introduction-overview and main body, are not completely under the discretion of the author, as he/she had to face the general criticism of all the other authors. Only the third part of the chapter is entirely for the authors' own personal contribution.

The Handbook meetings were generously financed by OUP, by SERC contract SO/809/86, by the Department of Computing at Imperial College, and by several anonymous private donations.

We would like to thank our colleagues, authors, second readers, and students for their effort and professionalism in producing the manuscripts for the Handbook. We would particularly like to thank the staff of OUP for their continued and enthusiastic support, and Mrs Jane Spurr, our OUP Administrator, for her dedication and efficiency.

London  
April 1994

S. Abramsky and D. M. Gabbay





# Contents

<b>List of contributors</b>	xv
-----------------------------	----

<b>Models for concurrency</b>	1
-------------------------------	---

*Glynn Winskel and Mogens Nielsen*

1	Introduction	1
2	Transition systems	7
2.1	A category of transition systems	7
2.2	Constructions on transition systems	10
3	A process language	20
3.1	Operational semantics (version 1)	22
3.2	Operational semantics (version 2)	23
3.3	An example	26
4	Synchronisation trees	28
5	Languages	32
6	Relating semantics	34
7	Trace languages	36
7.1	A category of trace languages	37
7.2	Constructions on trace languages	40
8	Event structures	41
8.1	A category of event structures	44
8.2	Domains of configurations	45
8.3	Event structures and trace languages	47
9	Petri nets	61
9.1	A category of Petri nets	65
9.2	Constructions on nets	66
10	Asynchronous transition systems	70
10.1	Asynchronous transition systems and trace languages	73
10.2	Asynchronous transition systems and nets	75
10.3	Properties of conditions	91
11	Semantics	102
11.1	Embeddings	102
11.2	Labelled structures	106
11.3	Operational semantics	109
12	Relating models	118
13	Notes	122
	Acknowledgements	128
	References	129

Appendix A: A basic category	135
Appendix B: Fibred categories	135
Appendix C: Operational semantics—proofs	139

## **Concrete process algebra** 149

*J. C. M. Baeten and C. Verhoef*

1	Introduction	150
2	Concrete sequential processes	152
2.1	Introduction	152
2.2	Basic process algebra	152
2.3	Recursion in BPA	167
2.4	Projection in BPA	169
2.5	Deadlock	181
2.6	Empty process	184
2.7	Renaming in BPA	190
2.8	The state operator	197
2.9	The extended state operator	201
2.10	The priority operator	204
2.11	Basic process algebra with iteration	217
2.12	Basic process algebra with discrete relative time	220
2.13	Basic process algebra with other features	224
2.14	Decidability and expressiveness results in BPA	226
3	Concrete concurrent processes	228
3.1	Introduction	229
3.2	Syntax and semantics of parallel processes	229
3.3	Extensions of PA	235
3.4	Extensions of $PA_{\delta}$	241
3.5	Syntax and semantics of communicating processes	244
3.6	Extensions of ACP	251
3.7	Decidability and expressiveness results in ACP	255
4	Further Reading	260
	References	260

## **Correspondence between operational and denotational semantics: the full abstraction problem for PCF**

*C.-H. L. Ong*

1	Introduction	270
1.1	Relating operational and denotational semantics	270
1.2	Full abstraction problem for PCF	274
1.3	Quest for a solution: a survey	276

1.4	Organization of the chapter	280
1.5	A selected bibliography	281
2	A programming language for computable functions	282
2.1	The programming language PCF	283
2.2	Syntax of the reduction system $\lambda_{\vec{Y}}(\mathbf{A})$	284
2.3	Reduction rules of the system $\lambda_{\vec{Y}}(\mathbf{A})$	287
2.4	Operational properties of $\lambda_{\vec{Y}}(\mathbf{A})$	288
2.5	Böhm trees and the Syntactic Continuity Theorem	292
2.6	Sequentiality and stability	295
2.7	The programming language PCF	299
3	Operational and denotational semantics of PCF	301
3.1	Context Lemma and observational extensionality	302
3.2	Denotational models	304
3.3	Adequacy	308
3.4	Order-extensional, continuous, fully abstract model	313
3.5	Full abstraction and non-full abstraction results	319
4	Towards a characterization of PCF-sequentiality	325
4.1	Concrete data structures and sequential functions	327
4.2	dI-domains and stable functions	335
4.3	Sequential algorithms	339
4.4	Observable algorithms and PCF with error values	342
4.5	Towards a characterization of sequentiality	345
	Acknowledgements	349
	References	350

## Effective algebras 357

*V. Stoltenberg-Hansen and J. V. Tucker*

1	Introduction	358
1.1	Computing in algebras	359
1.2	Examples of countable algebras	361
1.3	Examples of uncountable algebras	363
1.4	Definitions of computable algebras	364
1.5	General algebraic framework for computations on algebras	366
1.6	Historical notes on computable algebra	368
1.7	Objectives and structure of the chapter	371
1.8	Prerequisites	372
2	Computable algebras	372
2.1	Preliminaries on algebras	373
2.2	Computable, semicomputable, and cosemicomputable algebras	377
2.3	Invariance	387
2.4	A few computable constructions	397
2.5	Concluding remarks	402
3	Algebraic characterisations of computable algebras	402
3.1	Computable data types and their specification	403



3.2	Equational specifications	411
3.3	Adequacy theorem	421
3.4	Equational specifications and term rewriting	427
3.5	Proof of First Completeness Theorem	433
3.6	Concluding remarks	444
4	Domains and approximations for topological algebras	445
4.1	Approximation structures and topologies	446
4.2	Representing topological algebras by structured domains	451
4.3	Inverse limits and ultrametric algebras	456
4.4	Total elements in domains	467
4.5	Representability of locally compact Hausdorff algebras	472
5	Effective domains	479
5.1	Basic theory	480
5.2	Constructive subdomains	485
5.3	Algebras effectively approximable by domains	489
5.4	The Myhill–Shepherdson Theorem	501
5.5	The Kreisel–Lacombe–Shoenfield Theorem	505
	References	512

## **Abstract interpretation: a semantics-based tool for program analysis**

527

*Neil D. Jones and Flemming Nielson*

1	Introduction	528
1.1	Goals and motivations	528
1.2	Relation to program verification and transformation	535
1.3	The origins of abstract interpretation	535
1.4	A sampling of data-flow analyses	536
1.5	Outline	538
2	Basic concepts and problems to be solved	539
2.1	A naive analysis of the simple program	540
2.2	Accumulating semantics for imperative programs	542
2.3	Correctness and safety	548
2.4	Scott domains, lattice duality, and meet versus join	555
2.5	Abstract values viewed as relations or predicates	556
2.6	Important points from earlier sections	560
2.7	Towards generalizing the Cousot framework	561
2.8	Proving safety by logical relations	565
3	Abstract interpretation using a two-level metalanguage	568
3.1	Syntax of metalanguage	569
3.2	Specification of analyses	575
3.3	Correctness of analyses	588
3.4	Induced analyses	595
3.5	Expected forms of analyses	604
3.6	Extensions and limitations	609

## *Contents*

xiii

4	Other analyses, language properties, and language types	610
4.1	Approaches to abstract interpretation	611
4.2	Examples of instrumented semantics	614
4.3	Analysis of functional languages	616
4.4	Complex abstract values	621
4.5	Abstract interpretation of logic programs	623
5	Glossary	627
	References	629

## **Index**

637





# Contributors

**J. C. M. Baeten** Department of Computer Science, Eindhoven University of Technology, PO Box 513, 5600 MB Eindhoven, The Netherlands

**Neil D. Jones** Institute of Datalogy, University of Copenhagen, DIKU, Universitetsparken 1, DK 2100 Copenhagen Ø, Denmark

**Mogens Nielsen** Aarhus University, Computer Science Department, Ny Munkegade 116, DK 8000 Aarhus C, Denmark

**Flemming Nielson** Department of Computer Science, University of Aarhus, Building 540, Ny Munkegade 116, DK 8000 Aarhus C, Denmark

**C.-H. L. Ong** Programming Research Group, Oxford University Computing Laboratory, 11 Keble Road, Oxford, OX1 3QD, UK

**V. Stoltenberg-Hansen** Department of Mathematics, Uppsala Universitet, Box 480, S-751 06 Uppsala, Sweden

**J. V. Tucker** Department of Computer Science, University College Swansea, Singleton Park, Swansea, SA2 8PP, UK

**C. Verhoef** Faculty of Mathematics and Computer Science, University of Amsterdam, Kruislaan 403, NL-1098 SJ Amsterdam, The Netherlands

**Glynn Winskel** Aarhus University, Computer Science Department, Ny Munkegade 116, DK 8000 Aarhus C, Denmark



# Models for concurrency

Glynn Winskel and Mogens Nielsen

---

## Contents

1	Introduction . . . . .	2
2	Transition systems . . . . .	7
	2.1 A category of transition systems . . . . .	7
	2.2 Constructions on transition systems . . . . .	10
3	A process language . . . . .	20
	3.1 Operational semantics (version 1) . . . . .	22
	3.2 Operational semantics (version 2) . . . . .	23
	3.3 An example . . . . .	26
4	Synchronisation trees . . . . .	28
5	Languages . . . . .	32
6	Relating semantics . . . . .	34
7	Trace languages . . . . .	36
	7.1 A category of trace languages . . . . .	37
	7.2 Constructions on trace languages . . . . .	40
8	Event structures . . . . .	41
	8.1 A category of event structures . . . . .	44
	8.2 Domains of configurations . . . . .	45
	8.3 Event structures and trace languages . . . . .	47
9	Petri nets . . . . .	61
	9.1 A category of Petri nets . . . . .	65
	9.2 Constructions on nets . . . . .	66
10	Asynchronous transition systems . . . . .	70
	10.1 Asynchronous transition systems and trace languages . . . . .	73
	10.2 Asynchronous transition systems and nets . . . . .	75
	10.3 Properties of conditions . . . . .	91
11	Semantics . . . . .	102
	11.1 Embeddings . . . . .	102
	11.2 Labelled structures . . . . .	106
	11.3 Operational semantics . . . . .	109
12	Relating models . . . . .	118
13	Notes . . . . .	122

A	A basic category . . . . .	135
B	Fibred categories . . . . .	135
C	Operational semantics—proofs . . . . .	139

## 1 Introduction

The purpose of this chapter is to provide a survey of the fundamental models for distributed computations used and studied within theoretical computer science. Such models have the nature of mathematical formalisms in which to describe and reason about the behaviour of distributed computational systems. Their purpose is to provide an understanding of systems and their behaviour in theory, and to contribute to methods of design and analysis in practice.

In the rich theory of sequential computational systems, several mathematical models have been studied in depth, *e.g.* Turing machines, lambda calculus, Post systems, Markov systems, random access machines, *etc.* A main result of this theory is that the formalisms are all equivalent, in the sense that their behaviours in terms of input-output functions are the same.

However, in reality, few computational systems are sequential. On all levels, from a small chip to a world-wide network, computational behaviours are often distributed, in the sense that they may be seen as spatially separated activities accomplishing a joint task. Many such systems are not meant to terminate, and hence it makes little sense to talk about their behaviours in terms of traditional input-output functions. Rather, we are interested in the behaviour of such systems in terms of the often complex patterns of stimuli/response relationships varying over time. For this reason such systems are often referred to as *reactive systems*.

In the study of reactive systems, we are forced to take a different, less abstract, view of behaviours than the traditional one equating behaviour with an input-output function. A notion of behaviour is needed which expresses the patterns of actions which a system can perform, so as to capture such aspects as deadlock, mutual exclusion, starvation, *etc.*

One may see such models as providing a foundation for the development of all other theoretical and practical research areas on distributed computing. To give some examples, the models are used to provide the semantics of process description languages, and hence the basis of the many behavioural equivalences studied in the literature on process calculi. They are used to give the formal definition of specification logics, and hence underpin the work on verification of systems with respect to such specifications. And given this, they are at the heart of the development of automated tools for reasoning about distributed systems.

Numerous models have been suggested and studied over the past 10–15 years. Here we shall not attempt to present a complete survey. Rather, we



have chosen to present in fair detail a few key models.

Common to all the models we consider, is that they rest on the central idea of atomic actions, over which the behaviour of a system is defined. The models differ mainly with respect to what behavioural features of systems are represented. Some models are more abstract than others, and this fact is often used in informal classifications of the models with respect to expressibility. One of our aims is to present principal representatives of models, covering the landscape from the most abstract to the most concrete, and to formalise the nature of their relationships by explicitly representing the steps of abstraction that are involved in moving between them. In other words we would like to set the scene for a formal classification of models.

Let us be more specific. Imagine a very simple distributed computational system consisting of three individual components, each performing some independent computations, involving one (Sender) occasionally sending messages to another (Receiver) via the third (Medium):



Imagine modelling the behaviour of this system in terms of some atomic actions of the individual components, and the two actions of delivering a message from Sender to Medium, and passing on a message from Medium to Receiver. Obviously, having fixed such a set of atomic actions, we have also fixed a particular physical level at which to model our system.

One main distinction in the classification of models is that between interleaving and noninterleaving models. The main characteristic of an interleaving model is that it abstracts away from the fact that our system is actually composed of three independently computing agents, and models the behaviour in terms of purely sequential patterns of actions. Formally, the behaviour of our system will be expressed in terms of the nondeterministic merging, or interleaving, of the sequential behaviours of the three components. Prominent examples of such models are transition systems [Keller, 1976], synchronisation trees [Milner, 1980], acceptance trees [Hennessy, 1988], and Hoare traces [Hoare, 1981].

It is important to realise that in many situations abstraction like this is exactly what is wanted, and it has been demonstrated in the references above that many interesting and important properties of distributed systems may be expressed and proved based on interleaving models. The whole point of abstraction is, of course, to ignore aspects of the system which are irrelevant for the features we would like to reason about.

However, there may be situations in which it is important to keep the information that our system is composed of the three independently computing components, a possibility offered by the so-called noninterleaving models, with Petri nets [Adv, 1987], event structures [Winskel, 1987a], and Mazurkiewicz traces [Mazurkiewicz, 1988] as prime examples. One

such situation is that where some behavioural properties (typically liveness properties) rest on the fact that each component is a separate physical entity independently making its own computational progress. Dealing with such properties in interleaving models is often handled by a specific naming of the actions belonging to the components combined with logical assertions expressing progress assumptions for the system under study, *i.e.* handled outside the model in an *ad hoc* fashion.

Another issue is how models deal with the concept of nondeterminism in computations, distinguishing between so-called linear-time and branching-time models.

Imagine that in our system the component Medium is erroneous, in the sense that delivering a message from the Sender may leave Medium in either a normal state, having accepted the message and ready for another delivery or a passing of a message to Receiver, or a faulty state insisting on another delivery. A linear-time model will abstract away from this possibility of a suspended behaviour of the process Medium (and hence from some possibilities of deadlock). These models typically express the full nondeterministic behaviour of a system in terms of its set of possible (determinate) “runs” (or computation paths). Major examples of the structures used to model runs are Hoare traces, Mazurkiewicz traces and Pratt’s pomsets.

As indicated, in many situations a more detailed representation of when nondeterministic choices are made during a computation is necessary to reflect absence of deadlocks and other safety properties of systems. This is possible to various degrees in branching-time models like synchronisation and acceptance trees, Petri nets, and event structures. Of course, the treatment of nondeterminism is particularly important for the interleaving models, where parallel activities are also expressed in terms of nondeterminism.

Finally, yet a third distinction is made between those models allowing an explicit representation of the (possibly repeating) states in a system, and models abstracting away from such information, which focus instead on the behaviour in terms of patterns of occurrences of actions over time. Prime examples of the first type are transition systems and Petri nets, and of the second type, trees, event structures, and traces.

Thus the seemingly confusing world of models for concurrency can be structured according to a classification based on the expressiveness of the various models. In following through this programme, category theory is a convenient language for formalising the relationships between models.

To give an idea of the role categories play, let’s focus attention on transition systems as a model of parallel computation. A transition system consists of a set of states with labelled transitions between them. Assume the transition system has a distinguished initial state so that it can be presented by

$$(S, i, L, \text{Tran})$$

where  $S$  is a set of states with initial state  $i$ ,  $L$  is a set of labels, and the transitions elements of  $Tran \subseteq S \times L \times S$ ; a transition  $(s, a, s')$  is generally written as  $s \xrightarrow{a} s'$ . It then models a process whose transitions represent the process's atomic actions while the labels are action names; starting from the initial state, it traces out a computation path as transitions occur consecutively.

Processes often arise in relationship to other processes. For example, one process may refine another, or perhaps one process is a component of another. The corresponding relationships between behaviours are often expressed as morphisms between transition systems. For several models, there is some choice in how to define appropriate morphisms—it depends on the extent of the relationship between processes we wish to express. But here, we have an eye to languages like CCS, where communication is based on the synchronisation of atomic actions. From this viewpoint, we get a useful class of morphisms, sufficient to relate the behaviour of processes and their subcomponents, by taking a morphism from one transition system  $T$  to another  $T'$  to be a pair  $(\sigma, \lambda)$ , in which

- $\sigma$  is a function from the states of  $T$  to those of  $T'$  that sends the initial state of  $T$  to that of  $T'$ ,
- $\lambda$  is a *partial* function from the labels of  $T$  to those of  $T'$  such that for any transition  $s \xrightarrow{a} a'$  of  $T$ , if  $\lambda(a)$  is defined, then  $\sigma(s) \xrightarrow{\lambda(a)} \sigma(s')$  is a transition of  $T'$ ; otherwise, if  $\lambda(a)$  is undefined, then  $\sigma(s) = \sigma(s')$ .

Morphisms respect a choice of granularity for actions in the sense that an action may only be sent to at most one action, and not to a computation consisting of several actions. By taking  $\lambda$  to be a *partial* function on labels, we in particular accommodate the fact that projecting from a parallel composition of processes (*e.g.* in CCS) to a component may not only change action names, but also allow some actions to vanish if they do not correspond to those of the component, in which case their occurrence has no effect on the state of the component.

This definition of morphism is sufficient to express the relationship between a constructed process and its components as morphisms, at least within a language like CCS. But conversely the choice of morphisms also produces constructions. This is because transition systems and their morphisms form a category, and universal constructions (including limits and colimits) of a category are determined uniquely to within isomorphism, once they exist. In fact the universal constructions of the category of transition systems form the basis of a process description language. It is a little richer than that of CCS and CSP in the sense that their operations are straightforwardly definable within it.

When we consider other models as categories the same universal constructions yield sensible interpretations of the process-language constructs. Without categories this unity is lost; indeed, the denotations of parallel



compositions, often nontrivial to define, have been invented in an *ad hoc* fashion for most of the models we present.

Categorical notions also come into play in relating different models. Another model, synchronisation trees, arises by ignoring repetitive behaviour. We can identify synchronisation trees with special transition systems (those with no loops, no distinct transitions to the same state, in which all states are reachable). Synchronisation trees inherit morphisms from transition systems, and themselves form a category. The inclusion of synchronisation trees in transition systems is a functor. But more, the inclusion functor is part of an adjunction; the inclusion functor (the left adjoint) is accompanied, in a uniquely determined way, by a functor (the right adjoint) unfolding transition systems to synchronisation trees. A further step of abstraction, this time ignoring the branching of computation paths, takes us to languages as models of processes. A process is represented by the set of strings of actions it can perform. Languages can be identified with certain kinds of synchronisation trees and again this inclusion is part of an adjunction.

$$\begin{array}{ccccc} \text{Languages} & \hookrightarrow & \text{Synchronisation} & \hookrightarrow & \text{Transition} \\ & & \text{trees} & & \text{systems} \end{array}$$

As parts of adjunctions the functors enjoy preservation properties, which, coupled with the understanding of process operations as universal constructions, are useful in relating different semantics.

Here we have discussed just the three simplest models, but the same general approach applies to other models. The main idea is that each model will be equipped with a notion of morphism, making it into a category in which the operations of process calculi are universal constructions. The morphisms will preserve behaviour, at the same time respecting a choice of granularity of actions in the description of processes. One role of the morphisms is to relate the behaviour of a construction on processes to that of its components. As we shall see, it turns out that certain kinds of adjunctions (reflections and coreflections<sup>1</sup>) provide a way to express that one model is embedded in (is more abstract than) another, even when the two models are expressed in very different mathematical terms. One adjoint will say how to embed the more abstract model in the other, the other will abstract away from some aspect of the representation, in the same manner as has been described above. The adjunctions not only provide an aid in the understanding of the different models and their relationships, but are also a vehicle for the transfer of techniques from one model to another. In this chapter we concentrate on the role of models in giving a formal semantics to process description languages. The understanding of their operations as universal constructions guides us away from *ad hoc* definitions. And,

---

<sup>1</sup>A *reflection* is an adjunction in which the right adjoint is full and faithful, a *coreflection* one where the left adjoint is full and faithful.

importantly, we can use the preservation properties of adjoints to show how a semantics in one model translates to a semantics in another.

In summary, our goal is to survey a few, fundamental models for concurrency, and exploit category theory as a language for describing their structure and their relationships.<sup>2</sup>

## 2 Transition systems

Transition systems are a commonly used and understood model of computation. They provide the basic operational semantics for Milner's Calculus of Communicating Systems (CCS) and often underlie other approaches, such as that of Hoare's Communicating Sequential Processes (CSP). The constructions on transition systems used in such methods can frequently be seen as universal in a category of transition systems, where the morphisms can be understood as expressing the partial simulation (or refinement) of one process by another. By "abstract nonsense" the universal properties will characterise the constructions to within isomorphism. More strikingly, the same universal properties will apply in the case of other models like Petri nets or event structures, which are seemingly very different in nature.

### 2.1 A category of transition systems

Transition systems consist of a set of states, with an initial state, together with transitions between states which are labelled to specify the kind of events they represent.

**Definition 2.1.1.** A *transition system* is a structure

$$(S, i, L, \text{Tran})$$

where

- $S$  is a set of *states* with *initial state*  $i$ ,
- $L$  is a set of *labels*, and
- $\text{Tran} \subseteq S \times L \times S$  is the *transition relation*.

This definition narrows attention to transition systems, which are *extensional*: no two distinct transitions with the same label have the same pre and post states.

**Notation 2.1.2.** Let  $(S, i, L, \text{Tran})$  be a transition system. We write

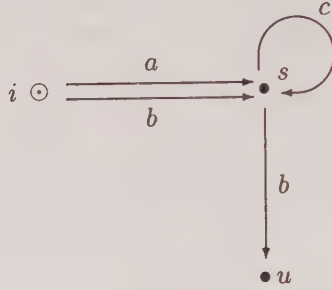
$$s \xrightarrow{a} s'$$

---

<sup>2</sup>A knowledge of basic category theory, up to an acquaintance with the notion of adjunction, is sufficient for the whole chapter. However, a light acquaintance, and some goodwill, should suffice, at least for the earlier parts. Good introductory references are [Peirce, 1991] and [Barr and Wells, 1990], while [MacLane, 1971] remains the classic text.



to indicate that  $(s, a, s') \in \text{Tran}$ . This notation lends itself to the familiar graphical notation for transition systems. For example,



represents a transition system which at the initial state  $i$  (encircled to distinguish it) can perform either an  $a$  or a  $b$  transition to enter the state  $s$  at which it can repeatedly perform a  $c$  transition or a  $b$  transition to enter state  $u$ .

We occasionally write

$$s \not\stackrel{a}{\rightarrow}$$

to mean there is no transition  $(s, a, s')$ . It is sometimes convenient to extend the arc-notation to strings of labels and write

$$s \xrightarrow{v} s',$$

when  $v = a_1 a_2 \cdots a_n$  is a, possibly empty, string of labels in  $L$ , to mean

$$s \xrightarrow{a_1} s_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} s_n,$$

for some states  $s_1, \dots, s_n$ . A state  $s$  is said to be *reachable* when  $i \xrightarrow{v} s$  for some string  $v$ .

**Definition 2.1.3.** Say a transition system  $T = (S, i, L, \text{Tran})$  is *reachable* iff every state in  $S$  is reachable from  $i$  and for every label  $a$  there is a transition  $(s, a, s') \in \text{Tran}$ . Say  $T$  is *acyclic* iff, for all strings of labels  $v$ , if  $s \xrightarrow{v} s$  then  $v$  is empty.

It is convenient to introduce *idle* transitions, associated with any state. This has to do with our representation of partial functions, explained in Appendix A. We view a partial function from a set  $L$  to a set  $L'$  as a (total) function  $\lambda : L \rightarrow L' \cup \{*\}$ , where  $*$  is a distinguished element standing for “undefined”. This representation is reflected in our notation  $\lambda : L \rightarrow_* L'$  for a partial function  $\lambda$  from  $L$  to  $L'$ . It assumes that  $*$  does not appear

in the sets  $L$  and  $L'$ , and more generally we shall assume that the reserved element  $*$  does not appear in any of the sets appearing in our constructions.

**Definition 2.1.4.** Let  $T = (S, i, L, Tran)$  be a transition system. An *idle transition* of  $T$  typically consists of  $(s, *, s)$ , where  $s \in S$ . Define

$$Tran_* = Tran \cup \{(s, *, s) \mid s \in S\}.$$

Idle transitions help simplify the definition of morphisms between transition systems. Morphisms on transition systems have already been discussed in the Introduction. There, a morphism  $T \rightarrow T'$  between transition systems was presented as consisting of a pair, one component  $\sigma$  being a function on states, preserving initial states, and the other a partial function  $\lambda$  on labels with the property that together they send a transition of  $T$  to a transition of  $T'$ , whenever this makes sense. More precisely, if  $(s, a, s')$  is a transition of  $T$  then  $(\sigma(s), \lambda(a), \sigma(s'))$  is a transition of  $T'$  provided  $\lambda(a)$  is defined; otherwise, in the case where  $\lambda(a)$  is undefined, it is insisted that the two states  $\sigma(s)$  and  $\sigma(s')$  are equal. With the device of idle transitions and the particular representation of partial functions, the same effect is achieved with the following definition:

**Definition 2.1.5.** Let

$$T_0 = (S_0, i_0, L_0, Tran_0) \text{ and}$$

$$T_1 = (S_1, i_1, L_1, Tran_1)$$

be transition systems. A *morphism*  $f : T_0 \rightarrow T_1$  is a pair  $f = (\sigma, \lambda)$  where

- $\sigma : S_0 \rightarrow S_1$
- $\lambda : L_0 \rightarrow_* L_1$  are such that  $\sigma(i_0) = i_1$  and

$$(s, a, s') \in Tran_0 \Rightarrow (\sigma(s), \lambda(a), \sigma(s')) \in Tran_{1*}.$$

The intention behind the definition of morphism is that the effect of a transition with label  $a$  in  $T_0$  leads to inaction in  $T_1$  precisely when  $\lambda(a)$  is undefined. In our definition of morphism, idle transitions represent this inaction, so we avoid the fuss of considering whether or not  $\lambda(a)$  is defined. With the introduction of idle transitions, morphisms on transition systems can be described as preserving transitions and the initial state. It is stressed that an idle transition  $(s, *, s)$  represents inaction, and is to be distinguished from the action expressed by a transition  $(s, a, s')$  for a label  $a$ .

Morphisms preserve initial states and transitions and so clearly preserve reachable states:

**Proposition 2.1.6.** Let  $(\sigma, \lambda) : T_0 \rightarrow T_1$  be a morphism of transition systems. Then if  $s$  is a reachable state of  $T_0$  then  $\sigma(s)$  is a reachable state of  $T_1$ .

Transition systems and their morphisms form a category which will be the first important category in our study:

**Proposition 2.1.7.** *Taking*

- *the class of objects to be transition systems,*
- *the class of morphisms to be those of transition systems,*

*defines a category, where*

- *the composition of two morphisms  $f = (\sigma, \lambda) : T_0 \rightarrow T_1$  and  $g = (\sigma', \lambda') : T_1 \rightarrow T_2$  is  $g \circ f = (\sigma' \circ \sigma, \lambda' \circ \lambda) : T_0 \rightarrow T_2$ —here composition on the left of a pair is that of total functions while that on the right is of partial functions, and*
- *the identity morphism for a transition system  $T$  has the form  $(1_S, 1_L)$ , where  $1_S$  is the identity function on states  $S$  and  $1_L$  is the identity function on the labelling set  $L$  of  $T$ .*

**Proof.** It is easily checked that composition is associative and has the identities claimed. ■

**Definition 2.1.8.** Denote by **T** the category of labelled transition systems given by the last proposition.

## 2.2 Constructions on transition systems

Transition systems are used in many areas. We focus on their use in modelling process calculi. The constructions used there can be understood as universal constructions in the category of transition systems. The point is not to explain the familiar in terms of the unfamiliar, but rather to find characterisations of sufficient generality that they apply to the other models as well. As we will see, the category of transition systems is rich in categorical constructions which furnish the basic combinators for a language of parallel processes.

### 2.2.1 Restriction

Restriction is an important operation on processes. For example, in Milner's CCS, labels are used to distinguish between input and output to channels, connected to processes at ports, and internal events. The effect of hiding all but a specified set of ports of a process, so that communication can no longer take place at the hidden ports, is to restrict the original behaviour of the process to transitions which do not occur at the hidden ports. Given a transition system and a subset of its labelling set, the operation of restriction removes all transitions whose labels are not in that set:

**Definition 2.2.1.** Let  $T' = (S, i, L', \text{Tran}') be a transition system. Assume  $L \subseteq L'$  and let  $\lambda : L \hookrightarrow L'$  be the associated inclusion morphism,$

taking  $a$  in  $L$  to  $a$  in  $L'$ . Define the *restriction*  $T' \upharpoonright \lambda$  to be the transition system  $(S, i, L, \text{Tran})$  with

$$\text{Tran} = \{(s, a, t) \in \text{Tran}' \mid a \in L\}.$$

Restriction is a construction which depends on labelling sets and functions between them. Seeing it as a categorical construction involves dealing explicitly with functions on labelling sets and borrowing a fundamental idea from fibred category theory. We observe that there is a functor  $p : \mathbf{T} \rightarrow \mathbf{Set}_*$ , to the category of sets with partial functions, which sends a morphism of transition systems  $(\sigma, \lambda) : T \rightarrow T'$  between transition systems  $T$  over  $L$  and  $T'$  over  $L'$  to the partial function  $\lambda : L \rightarrow_* L'$ . Associated with a restriction  $T' \upharpoonright L$  is a morphism  $f : T' \upharpoonright L \rightarrow T'$ , given by  $f = (1_S, \lambda)$  where  $\lambda$  is the inclusion map  $\lambda : L \hookrightarrow L'$ . In fact the morphism  $f$  is essentially an “inclusion” of the restricted into the original transition system. The morphism  $f$  associated with the restriction has the universal property that:

For any  $g : T \rightarrow T'$  a morphism in  $\mathbf{T}$  such that  $p(g) = \lambda$  there is a unique morphism  $h : T \rightarrow T' \upharpoonright L$  such that  $p(h) = 1_L$  and  $f \circ h = g$ . In a diagram:

$$\begin{array}{ccc} & T & \\ & \downarrow h & \searrow g \\ T' \upharpoonright L & \xrightarrow{f} & T' \\ p \downarrow & & \\ \mathbf{T} & & L \xrightarrow{\lambda} L' \\ \downarrow & & \\ \mathbf{Set}_* & & \end{array}$$

This says that the “inclusion” morphisms associated with restrictions are *cartesian*; the morphism  $f$  is said to be a *cartesian lifting* of  $\lambda$  with respect to  $T'$ . In fact, they are *strong cartesian*—see Appendix B.

**Proposition 2.2.2.** *Let  $\lambda : L \rightarrow_* L'$  be an inclusion. Let  $T'$  be a labelled transition system, with states  $S$ . There is a morphism  $f : T' \upharpoonright L \rightarrow T'$ , given by  $f = (1_S, \lambda)$ . It is (strong) cartesian.*

Because an inclusion  $\lambda : L \hookrightarrow L'$  has cartesian liftings for any  $T'$  with labelling set  $L'$ , restriction automatically extends to a functor from transition systems with labelling set  $L'$  to those with labelling set  $L$ . To state this more fully, note first that a labelling set  $L$  is associated with a subcategory of transition systems  $p^{-1}(L)$ , called the *fibre* over  $L$ , consisting of objects those transition systems  $T$  for which  $p(T) = L$  (i.e. whose labelling set is  $L$ ) and morphisms  $h$  for which  $p(h) = 1_L$  (i.e. which preserve labels). An explicit choice of cartesian lifting for each  $T'$  in  $p^{-1}(L')$  (as

is provided by the restriction operation) yields a functor between fibres  $p^{-1}(L') \rightarrow p^{-1}(L)$ —the functor's action on morphisms coming from the universal property of cartesian liftings.

**Remark:** In fact there are (strong) cartesian liftings for any  $\lambda : L \rightarrow_* L'$  and any  $T'$  with labelling set  $L'$ , and the functor  $p : \mathbf{T} \rightarrow \mathbf{Set}_*$  is a fibration.

### 2.2.2 Relabelling

In CCS, one can make copies of a process by renaming its port names. This is associated with the operation of relabelling the transitions in the transition system representing its behaviour. When  $\lambda : L \rightarrow_* L'$  is total, the relabelling construction takes a transition system  $T$  with labelling set  $L$  to  $T\{\lambda\}$ , the same underlying transition system but relabelled according to  $\lambda$ .

**Definition 2.2.3.** Let  $T = (S, i, L, \text{Tran})$  be a transition system. Let  $\lambda : L \rightarrow L'$  be a total function. Define the *relabelling*  $T\{\lambda\}$  to be the transition system  $(S, i, L', \text{Tran}')$  where

$$\text{Tran}' = \{(s, \lambda(a), s') \mid (s, a, s') \in \text{Tran}\}.$$

The operation of relabelling is associated with a construction dual to that of cartesian lifting: that of forming a *cocartesian lifting*. Letting the transition system  $T$  have states  $S$ , there is a morphism  $f = (1_S, \lambda) : T \rightarrow T\{\lambda\}$ . Such a morphism is a cocartesian lifting of  $\lambda$  in the sense that:

For any  $g : T \rightarrow T'$  a morphism in  $\mathbf{T}$  such that  $p(g) = \lambda$  there is a unique morphism  $h : T\{\lambda\} \rightarrow T'$  such that  $p(h) = 1_{L'}$  and  $h \circ f = g$ . In a diagram:

$$\begin{array}{ccc} & & T' \\ & \nearrow g & \uparrow h \\ T & \xrightarrow{f} & T\{\lambda\} \\ \downarrow p & & \downarrow p \\ \mathbf{T} & & \mathbf{Set}_* \\ & \xrightarrow{\lambda} & L' \end{array}$$

**Proposition 2.2.4.** Let  $\lambda : L \rightarrow L'$  be a total function. Let  $T$  be a labelled transition system, with states  $S$ . There is a morphism  $f : T \rightarrow T\{\lambda\}$ , given by  $f = (1_S, \lambda)$  which is (strong) cocartesian—see Appendix B.

Relabelling extends to a functor  $p^{-1}(L) \rightarrow p^{-1}(L')$ , where  $\lambda : L \rightarrow L'$  is a total function.

**Remark:** The relabelling construction can also be defined more generally



when  $\lambda$  is partial. In fact there are (strong) cocartesian liftings for any  $\lambda : L \rightarrow_* L'$  and any  $T$  with labelling set  $L$ , and the functor  $p : \mathbf{T} \rightarrow \mathbf{Set}_*$  is a cofibration. Being a fibration too, this makes  $p$  a bifibration.

### 2.2.3 Product

Parallel compositions are central operations in process calculi; they set processes in communication with each other. Communication is via actions of mutual synchronisation, possibly with the exchange of values. Precisely how actions synchronise with each other varies enormously from one language to another, but for example in CCS and Occam processes are imagined to communicate over channels linking their ports. In these languages, an input action to a channel from one process can combine with an output action to the same channel from the other to form an action of synchronisation. The languages also allow for processes in a parallel composition to reserve the possibility of communicating with a, yet undetermined, process in the environment of both.

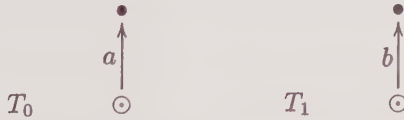
Parallel compositions in general can be derived, with restriction and relabelling, from a product operation on transition systems. In itself the product operation is a special kind of parallel composition in which all conceivable synchronisations are allowed.

**Definition 2.2.5.** Assume transition systems  $T_0 = (S_0, i_0, L_0, Tran_0)$  and  $T_1 = (S_1, i_1, L_1, Tran_1)$ . Their *product*  $T_0 \times T_1$  is  $(S, i, L, Tran)$  where

- $S = S_0 \times S_1$ , with  $i = (i_0, i_1)$ , and projections  $\rho_0 : S_0 \times S_1 \rightarrow S_0$ ,  $\rho_1 : S_0 \times S_1 \rightarrow S_1$ ,
- $L = L_0 \times_* L_1 = \{(a, *) \mid a \in L_0\} \cup \{(*, b) \mid b \in L_1\} \cup \{(a, b) \mid a \in L_0, b \in L_1\}$ , with projections  $\pi_0, \pi_1$ , and
- $(s, a, s') \in Tran_* \Leftrightarrow (\rho_0(s), \pi_0(a), \rho_0(s')) \in Tran_{0*} \ \& \ (\rho_1(s), \pi_1(a), \rho_1(s')) \in Tran_{1*}$ .

Define  $\Pi_0 = (\rho_0, \pi_0)$  and  $\Pi_1 = (\rho_1, \pi_1)$ .

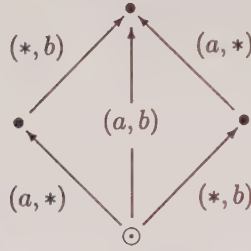
**Example 2.2.6.** Let  $T_0$  and  $T_1$  be the following transition systems



where  $T_0$  has  $\{a\}$  and  $T_1$  has  $\{b\}$  as the labelling set. The product of these labelling sets is

$$\{a\} \times_* \{b\} = \{(a, *), (a, b), (*, b)\}$$

with projections  $\lambda_0$  onto the first coordinate and  $\lambda_1$  onto the second. Thus  $\lambda_0(a, *) = \lambda_0(a, b) = a$  and  $\lambda_0(*, b) = *$ . Their product takes the form:



Intuitively, transitions with labels of the form  $(a, b)$  represent synchronisations between two processes set in parallel, while those labelled  $(a, *)$  or  $(*, b)$  involve only one process, performing the transition unsynchronised with the other. Clearly, this is far too “generous” a parallel composition to be useful as it stands, allowing as it does all possible synchronisations and absences of synchronisations between two processes. However, a wide range of familiar and useful parallel compositions can be obtained from the product operation by further applications of restriction (to remove unwanted synchronisations) and relabelling (to rename the results of synchronisations).

The product of transition systems  $T_0, T_1$  has projection morphisms  $\Pi_0 = (\rho_0, \pi_0) : T_0 \times T_1 \rightarrow T_0$  and  $\Pi_1 = (\rho_1, \pi_1) : T_0 \times T_1 \rightarrow T_1$ . They together satisfy the universal property required of a *product in a category*; viz. given any morphisms  $f_0 : T \rightarrow T_0$  and  $f_1 : T \rightarrow T_1$  from a transition system  $T$  there is a unique morphism  $h : T \rightarrow T_0 \times T_1$  such that  $\Pi_0 \circ h = f_0$  and  $\Pi_1 \circ h = f_1$ :

$$\begin{array}{ccccc}
 T_0 & \xleftarrow{\Pi_0} & T_0 \times T_1 & \xrightarrow{\Pi_1} & T_1 \\
 & \nwarrow f_2 & \uparrow h & \nearrow f_1 & \\
 & & T & & 
 \end{array}$$

**Proposition 2.2.7.** *Let  $T_0$  and  $T_1$  be transition systems. The construction  $T_0 \times T_1$  above, with projections  $\Pi_0 = (\rho_0, \pi_0)$ ,  $\Pi_1 = (\rho_1, \pi_1)$ , is a product in the category  $\mathbf{T}$ . A state  $s$  is reachable in  $T_0 \times T_1$  iff  $\rho_0(s)$  is reachable in  $T_0$  and  $\rho_1(s)$  is reachable in  $T_1$ .*

Although we have only considered binary products, all products exist in the category of transition systems. In particular, the empty product is the *nil* transition system

$$\mathit{nil} = (\{i\}, i, \emptyset, \emptyset),$$

consisting of a single initial state  $i$ . In this special case the universal property for products amounts to:

for any transition system  $T$ , there is a unique morphism  
 $h : T \rightarrow \mathit{nil}$ ;

that is,  $\mathit{nil}$  is a *terminal* object in the category of transition systems. The transition system  $\mathit{nil}$  is also an *initial* object in the category of transition systems:

for any transition system  $T$ , there is a unique morphism  
 $h : \mathit{nil} \rightarrow T$ .

We remark that the product-machine construction from automata theory arises as a *fibre product*, viz. a product in a fibre. Recall that a fibre  $p^{-1}(L)$  is a category which consists of the subcategory of transition systems with a common labelling set  $L$ , in which the morphisms preserve labels.

## 2.2.4 Parallel compositions

In the present framework, we do not obtain arbitrary parallel compositions as single universal constructions. Generally, they can be obtained from the product by restriction and relabelling; a parallel composition of  $T_0$  and  $T_1$ , with labelling sets  $L_0$ ,  $L_1$  respectively, is got by first taking their product, to give a transition system  $T_0 \times T_1$  with labelling set  $L_0 \times_* L_1$ , then restricting by taking  $(T_0 \times T_1) \upharpoonright S$  for an inclusion  $S \subseteq L_0 \times_* L_1$ , followed by a relabelling  $((T_0 \times T_1) \upharpoonright S)\{r\}$  with respect to a total function  $r : S \rightarrow L$ . In this way, using a combination of product, restriction, and relabelling we can represent all conceivable parallel compositions which occur by synchronisation.

In general parallel compositions are derived using a combination of product, restriction, and relabelling. We can present the range of associative, commutative parallel compositions based on synchronisation in a uniform way by using *synchronisation algebras*. A synchronisation algebra on a set  $L$  of labels (not containing the distinct elements  $*$ ,  $0$ ) consists of a binary, commutative, associative operation  $\bullet$  on  $L \cup \{*, 0\}$  such that

$$a \bullet 0 = 0 \text{ and } (a_0 \bullet a_1 = * \Leftrightarrow a_0 = a_1 = *)$$

for all  $a, a_0, a_1 \in L \cup \{*, 0\}$ . The role of  $0$  is to specify those synchronisations which are not allowed whereas the composition  $\bullet$  specifies a relabelling. (It may be helpful to look at the example ahead of the synchronisation algebra of CCS.) For a synchronisation algebra on labels  $L$ , let  $\lambda_0, \lambda_1 : L \times_* L \rightarrow_* L$  be the projections on its product in  $\mathbf{Set}_*$ . The parallel composition of two transition systems  $T_0, T_1$ , labelled over  $L$ , can be obtained as  $((T_0 \times T_1) \upharpoonright D)\{r\}$  where  $D \subseteq L \times_* L$  is the inclusion of

$$D = \{a \in L \times_* L \mid \lambda_0(a) \bullet \lambda_1(a) \neq 0\}$$

determined by the 0-element, and  $r : D \rightarrow L$  is the relabelling, given by

$$r(a) = \lambda_0(a) \bullet \lambda_1(a)$$

for  $a \in D$ .

We present two synchronisation algebras as examples, in the form of tables—more, including those for value-passing, can be found in [Winskel, 1982; Winskel, 1985].

**Example 2.2.8.** *The synchronisation algebra for pure CCS:* In CCS [Milner, 1989] events are labelled by  $a, b, \dots$  or by their complementary labels  $\bar{a}, \bar{b}, \dots$  or by the label  $\tau$ . The idea is that only two events bearing complementary labels may synchronise to form a synchronisation event labelled by  $\tau$ . Events labelled by  $\tau$  cannot synchronise further; in this sense they are invisible to processes in the environment, though their occurrence may lead to internal changes of state. All labelled events may occur asynchronously. Hence the synchronisation algebra for CCS takes the following form. The resultant parallel composition, of processes  $p$  and  $q$  say, is represented as  $p|q$  in CCS.

$\bullet$	$*$	$a$	$\bar{a}$	$b$	$\bar{b}$	$\dots$	$\tau$	0
$*$	$*$	$a$	$\bar{a}$	$b$	$\bar{b}$	$\dots$	$\tau$	0
$a$	$a$	0	$\tau$	0	0	$\dots$	0	0
$\bar{a}$	$\bar{a}$	$\tau$	0	0	0	$\dots$	0	0
$b$	$b$	0	0	0	$\tau$	$\dots$	0	0
$\cdot$	$\cdot$	$\cdot$	$\cdot$	$\cdot$	$\cdot$	$\dots$	$\cdot$	$\cdot$

**Example 2.2.9.** *The synchronisation algebra in TCSP:* In “theoretical” CSP—see [Hoare *et al.*, 1984; Brookes, 1985]—events are labelled by  $a, b, \dots$  or  $\tau$ . For one of its parallel compositions (usually written  $\parallel$ ) events must “synchronise on”  $a, b, \dots$ . In other words non- $\tau$ -labelled events cannot occur asynchronously. Rather, an  $a$ -labelled event in one component of a parallel composition must synchronise with an  $a$ -labelled event from the other component in order to occur; the two events must synchronise to form a synchronisation event again labelled by  $a$ . The synchronisation algebra for this parallel composition takes the following form:

$\bullet$	$*$	$a$	$b$	$\dots$	$\tau$	0
$*$	$*$	0	0	$\dots$	$\tau$	0
$a$	0	$a$	0	$\dots$	0	0
$b$	0	0	$b$	$\dots$	0	0
$\cdot$	$\cdot$	$\cdot$	$\cdot$	$\dots$	$\cdot$	$\cdot$



### 2.2.5 Sum

Nondeterministic sums in process calculi allow a process to be defined with the capabilities of two or more processes, so that the process can behave like one of several alternative processes. Which alternative can depend on what communications the environment offers, and in many cases, the nondeterministic sum plays an important role like that of the conditional of traditional sequential languages.

With the aim of understanding nondeterministic sums as universal constructions we examine coproducts in the category of transition systems.

**Definition 2.2.10.** Let  $T_0 = (S_0, i_0, L_0, \text{Tran}_0)$  and  $T_1 = (S_1, i_1, L_1, \text{Tran}_1)$  be transition systems. Define  $T_0 + T_1$  to be  $(S, i, L, \text{Tran})$  where

- $S = (S_0 \times \{i_1\}) \cup (\{i_0\} \times S_1)$  with  $i = (i_0, i_1)$ , and injections  $in_0, in_1$ ,
- $L = L_0 \uplus L_1$ , their disjoint union, with injections  $j_0, j_1$ ,
- transitions

$$t \in \text{Tran} \Leftrightarrow \exists (s, a, s') \in \text{Tran}_0. t = (in_0(s), j_0(a), in_0(s')) \text{ or} \\ \exists (s, a, s') \in \text{Tran}_1. t = (in_1(s), j_1(a), in_1(s')).$$

The construction  $T_0 + T_1$  on transition systems  $T_0, T_1$  has injection morphisms  $I_0 = (in_0, j_0) : T_0 \rightarrow T_0 + T_1$  and  $I_1 = (in_1, j_1) : T_1 \rightarrow T_0 + T_1$ . They together satisfy the universal property required of a *coproduct in a category*; viz. given any morphisms  $f_0 : T_0 \rightarrow T$  and  $f_1 : T_1 \rightarrow T$  to a transition system  $T$  there is a unique morphism  $h : T_0 + T_1 \rightarrow T$  such that  $h \circ I_0 = f_0$  and  $h \circ I_1 = f_1$ :

$$\begin{array}{ccccc} & & T & & \\ & f_0 \nearrow & \uparrow & \nwarrow f_1 & \\ T_0 & \xrightarrow{I_0} & T_0 + T_1 & \xleftarrow{I_1} & T_1 \end{array}$$

**Proposition 2.2.11.** Let  $T_0$  and  $T_1$  be transition systems. Then  $T_0 + T_1$ , with injections  $(in_0, j_0), (in_1, j_1)$ , is a coproduct in the category of transition systems.

A state  $s$  is reachable in a coproduct iff there is  $s_0$  reachable in  $T_0$  with  $s = in_0(s_0)$  or there is  $s_1$  reachable in  $T_1$  with  $s = in_1(s_1)$ .

The coproduct is not quite of the kind used in modelling the sum of CCS for example, because in the coproduct labels are made disjoint. We look to coproducts in a fibre.

For a labelling set  $L$ , each fibre  $p^{-1}(L)$  has coproducts. Recall that  $p^{-1}(L)$  is that subcategory of  $\mathbf{T}$  consisting of transition systems over a common labelling set  $L$  with those morphisms which project to the identity on  $L$ . In form fibre coproducts are very similar to coproducts of transition systems in general—they differ only in the labelling part.

**Definition 2.2.12.** Let  $T_0 = (S_0, i_0, L, \text{Tran}_0)$  and  $T_1 = (S_1, i_1, L, \text{Tran}_1)$  be transition systems over the same labelling set  $L$ . Define  $T_0 +_L T_1 = (S, i, L, \text{Tran})$  (note that it is over the same labelling set) where  $S = (S_0 \times \{i_1\}) \cup (\{i_0\} \times S_1)$  with  $i = (i_0, i_1)$ , and injections  $in_0, in_1$ , and

$$t \in \text{Tran} \Leftrightarrow \exists (s, a, s') \in \text{Tran}_0. t = (in_0(s), a, in_0(s')) \text{ or} \\ \exists (s, a, s') \in \text{Tran}_1. t = (in_1(s), a, in_1(s')).$$

**Proposition 2.2.13.** Let  $T_0$  and  $T_1$  be transition systems over  $L$ . The transition system  $T_0 +_L T_1$  with injections  $(in_0, 1_L)$  and  $(in_1, 1_L)$ , as defined above, is a coproduct in the subcategory of transition systems consisting of the fibre  $p^{-1}(L)$ .

Neither the coproduct or fibre coproduct of transition systems quite match the kind of sums used in modelling processes, for example, in CCS. The coproduct changes the labels, tagging them so they are disjoint, while the fibre coproduct, seemingly more appropriate because it leaves the labels unchanged, assumes that the transition systems have the same labelling set. A more traditional sum is the following:

**Definition 2.2.14.** Let  $T_0$  and  $T_1$  be transition systems over  $L_0$  and  $L_1$  respectively. Define their *sum*  $T_0 \oplus T_1$  to be  $(S, i, L_0 \cup L_1, \text{Tran})$  where  $S = (S_0 \times \{i_1\}) \cup (\{i_0\} \times S_1)$  with  $i = (i_0, i_1)$ , and injections  $in_0, in_1$ , and

$$t \in \text{Tran} \Leftrightarrow \exists (s, a, s') \in \text{Tran}_0. t = (in_0(s), a, in_0(s')) \text{ or} \\ \exists (s, a, s') \in \text{Tran}_1. t = (in_1(s), a, in_1(s')).$$

This sum can be understood as a fibre coproduct, but where we first form cocartesian liftings of the inclusion maps into the union of the labelling sets; this simply has the effect of enlarging the labelling sets to a common labelling set, their union, where we can form the fibre coproduct:

**Proposition 2.2.15.** Let  $T_0$  and  $T_1$  be transition systems over  $L_0$  and  $L_1$  respectively. Let  $j_k : L_k \hookrightarrow L_0 \cup L_1$  be the inclusion maps, for  $k = 0, 1$ . Then

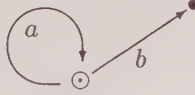
$$T_0 \oplus T_1 \cong T_0\{j_0\} +_{(L_0 \cup L_1)} T_1\{j_1\}.$$

Only coproducts of two transition systems have been considered. All coproducts exist in fibres and in the category of all transition systems. Thus there are indexed sums of transition systems of the kind used in CCS. The sum construction on transition systems is of the form required for CCS when the transition systems are “nonrestarting”, *i.e.* have no transitions back to the initial state. In giving and relating semantics we shall be mindful of this fact.

**Example 2.2.16.** The fibred coproduct  $T_0 +_L T_1$  of



both assumed to have the labelling set  $L = \{a, b\}$ , takes the form:



The sum can behave like  $T_0$  but then on returning to the initial state it behaves like  $T_1$ .

### 2.2.6 Prefixing

The categorical constructions form a basis for languages of parallel processes with constructs like parallel compositions and nondeterministic sums. The cartesian and cocartesian liftings give rise to restriction and relabelling operations as special cases, but the more general constructions, arising for morphisms in the base category which are truly partial, might also be useful constructions to introduce into a programming language. This raises an omission from our collection of constructions: we have not yet mentioned an operation which introduces new transitions from scratch. Traditionally, in languages like CCS, CSP, and Occam this is done with some form of prefixing operation, the effect of which is to produce a new process which behaves like a given process once a specified, initial action has taken place. Given a transition system, the operation of prefixing  $a(-)$  introduces a transition, with label  $a$ , from a new initial state to the former initial state in a copy of the transition system. One way to define prefixing on transition systems concretely is by:

**Definition 2.2.17.** Let  $a$  be a label (not  $*$ ). Define the *prefix*  $aT = (S', i', L', \text{Tran}')$  where

$$S' = \{\{s\} \mid s \in S\} \cup \{\emptyset\},$$

$$i' = \emptyset,$$

$$L' = L \cup \{a\},$$

$$\text{Tran}' = \{(\{s\}, b, \{s'\}) \mid (s, b, s') \in \text{Tran}\} \cup \{(\emptyset, a, \{i\})\}.$$

Because we do not ensure that the prefixing label is distinct from the former labels, prefixing does not extend to a functor on all morphisms of

transition systems. However, it extends to a functor on the subcategory of label-preserving morphisms, *i.e.* those morphisms  $(\sigma, \lambda) : T \rightarrow T'$  between transition systems for which  $\lambda : L \hookrightarrow L'$  is an inclusion function. As a special case, prefixing  $a(-)$  extends to a functor between fibres  $p^{-1}(L) \rightarrow p^{-1}(L \cup \{a\})$ .

### 3 A process language

A process language **Proc** and its semantics can be built around the constructions on the category of transition systems. Indeed the process language can be interpreted in all the models we consider. Its syntax is given by

$$t ::= nil \mid at \mid t_0 \oplus t_1 \mid t_0 \times t_1 \mid t \upharpoonright \Lambda \mid t\{\Xi\} \mid x \mid rec\ x.t$$

where  $a$  is a label,  $\Lambda$  is a subset of labels, and  $\Xi$  is a total function from labels to labels. We have seen how to interpret most of these constructions in transition systems, which in particular will yield a labelling set for each term. It is convenient to broaden the understanding of a restriction  $t \upharpoonright \Lambda$  so it means the same as  $t \upharpoonright \Lambda \cap L$  in the situation where the labelling set  $L$  does not include  $\Lambda$ . The denotation of  $t\{\Xi\}$  is obtained from the cocartesian lifting with respect to  $t$  of the function  $\Xi : L \rightarrow \Xi L$ , so that  $t$  with labelling set  $L$  is relabelled by  $\Xi$  cut down to domain  $L$ . The new construction is the recursive construction of the form  $rec\ x.t$ , involving  $x$ , a variable over processes. To smooth the presentation we insist that in a recursive definition  $rec\ x.t$  the occurrences of  $x$  in  $t$  are *guarded* in the sense that all occurrences of  $x$  in  $t$  lie within a prefix term.

The presence of process variables means that the denotation of a term as a transition system is given with respect to an environment  $\rho$  mapping process variables to transition systems. We can proceed routinely, by induction on the structure of terms, to give an interpretation of syntactic operations by those operations on transition systems we have introduced; for example, we set

$$\begin{aligned} \mathbf{T}[nil]\rho &= nil, \text{ for a choice of initial transition systems} \\ \mathbf{T}[t_0 \oplus t_1]\rho &= \mathbf{T}[t_0]\rho \oplus \mathbf{T}[t_1]\rho, \text{ the nondeterministic sum of} \\ &\text{section 2.2.5.} \end{aligned}$$

But how are we to interpret  $\mathbf{T}[rec\ x.t]\rho$ , for an environment  $\rho$ , assuming we have an interpretation  $\mathbf{T}[t]\rho'$  for any environment  $\rho'$ ?

There are several techniques in use for giving meaning to recursively defined processes and in this section we will discuss two. One approach is to use  $\omega$ -colimits with respect to some suitable subclass of morphisms in the category of transition systems and use the fact that the operations of the process language can be represented by functors which are continuous in the sense of preserving  $\omega$ -colimits. For example, all the operations needed to model **Proc** are continuous functors on the subcategory of transition

systems with label-preserving monomorphisms—this subcategory has all  $\omega$ -colimits. However, we can work more concretely and choose monomorphisms which are inclusions. In this instance the general method then becomes a mild generalisation of that of fixed points of continuous functions on a complete partial order. The method is based on the observation that transition systems almost form a complete partial order under the relation

$$(S, i, L, \text{tran}) \sqsubseteq (S', i', L', \text{tran}') \text{ iff } S \subseteq S' \ \& \ i = i' \ \& \ L \subseteq L' \ \& \ \text{tran} \subseteq \text{tran}'$$

associated with the existence of a morphism from one transition system to another based on inclusion of states and labelling sets. Objects of the category of transition systems do not form a set, but they do have least upper bounds of  $\omega$ -chains

$$T_0 \sqsubseteq T_1 \sqsubseteq \dots \sqsubseteq T_n \sqsubseteq \dots$$

of transition systems  $T_n = (S_n, i_n, L_n, \text{tran}_n)$ , for  $n \in \omega$ ; the least upper bound  $\bigcup_{n \in \omega} T_n$  is given simply by

$$\bigcup_{n \in \omega} T_n = \left( \bigcup_{n \in \omega} S_n, i_n, \bigcup_{n \in \omega} L_n, \bigcup_{n \in \omega} \text{tran}_n \right).$$

There is no unique least element, but rather a class of minimal transition systems  $(\{i\}, i, \emptyset, \emptyset)$  for a choice of initial state  $i$ . However, this is no obstacle to a treatment of guarded recursions based on the order  $\sqsubseteq$ .

First observe that each operation, prefixing, sum, product, restriction and relabelling, has been defined concretely, and in fact each operation is continuous with respect to  $\sqsubseteq$ . It follows that for a term  $t$ , and process variable  $x$ , the operation  $F$ , given by

$$F(T) = \mathbf{T}[[t]]\rho[T/x]$$

on transition systems  $T$ , is continuous. Moreover, if  $x$  is guarded in  $t$ , then for any choice of transition system  $T$ , the initial state of  $F(T)$  is the same,  $i$  say. Consequently, writing  $I = (\{i\}, i, \emptyset, \emptyset)$  we have

$$I \sqsubseteq F(I)$$

and inductively, by monotonicity,

$$I \sqsubseteq F(I) \sqsubseteq F^2(I) \sqsubseteq \dots \sqsubseteq F^n(I) \sqsubseteq \dots.$$

Write  $\text{fix}(F) =_{\text{def}} \bigcup_{n \in \omega} F^n(I)$ . By the continuity of  $F$  we see that  $\text{fix}(F)$  is the  $\sqsubseteq$ -least fixed point of  $F$ . In fact because  $F$  is defined from a term in which  $x$  is guarded, we can show a uniqueness property of its fixed points:



**Definition 3.0.1.** For a transition system  $T = (S, i, L, \text{tran})$ , define  $\mathcal{R}(T)$  to be the transition system  $(S', i, L', \text{Tran}')$  consisting of states  $S'$  reachable from  $i$ , with initial state  $i$ , and transitions  $\text{Tran}' = \text{Tran} \cap (S' \times L \times S')$  with labelling set  $L'$  consisting of those labels appearing in  $\text{Tran}'$ .

**Lemma 3.0.2.** *If  $T$  is a transition system for which  $T \cong \mathcal{R}(F(T))$ , a label-preserving isomorphism, then  $T \cong \mathcal{R}(\text{fix}(F))$ , a label-preserving isomorphism.*

**Proof.** The proof of this fact depends on several subsidiary definitions and results which we place in Appendix C. ■

We can now complete our denotational semantics, the denotation  $\mathbf{T}[\![\text{rec } x.t]\!]\rho$  being taken to be  $\text{fix}(F)$  where  $F(T) = \mathbf{T}[\![t]\!]\rho[T/x]$ .

### 3.1 Operational semantics (version 1)

Alternatively, we can give a structural operational semantics to our language on standard lines. In doing so it is useful to introduce a little notation concerning the combination of labels. For labels  $a, b$  define

$$a \times b = \begin{cases} * & \text{if } a = b = *, \\ (a, b) & \text{otherwise.} \end{cases}$$

This notation along with the use of idle transitions gives a single compact rule for product. The transitions between states, identified with closed terms, are given by the following rules:

$$\begin{array}{c} \frac{}{at \xrightarrow{a} t} \qquad \frac{}{t \xrightarrow{*} t} \\[10pt] \frac{t_0 \xrightarrow{a} t'_0}{t_0 \oplus t_1 \xrightarrow{a} t'_0} \quad a \neq * \qquad \frac{t_1 \xrightarrow{a} t'_1}{t_0 \oplus t_1 \xrightarrow{a} t'_1} \quad a \neq * \\[10pt] \frac{t_0 \xrightarrow{a} t'_0 \quad t_1 \xrightarrow{b} t'_1}{t_0 \times t_1 \xrightarrow{a \times b} t'_0 \times t'_1} \\[10pt] \frac{t \xrightarrow{a} t'}{t \upharpoonright \Lambda \xrightarrow{a} t \upharpoonright \Lambda} \quad a \in \Lambda \qquad \frac{t \xrightarrow{a} t'}{t\{\Xi\} \xrightarrow{\Xi(a)} t'\{\Xi\}} \\[10pt] \frac{t[\text{rec } x.t/x] \xrightarrow{a} t'}{\text{rec } x.t \xrightarrow{a} t'} \quad a \neq *. \end{array}$$

A closed term  $t$  determines a transition system with initial state  $t$  consisting of all states and transitions which are reachable from  $t$ .

Unfortunately the relationship between the transition systems obtained denotationally and operationally is a little obscure. There are several mismatches. One is that the categorical sum makes states of the two components of a sum disjoint, a property which cannot be shared by the transition system of the operational semantics, essentially because of incidental identifications of syntax. Furthermore, the transition system for recursive processes can lead to transition systems with transitions back to the initial state. As we have seen this causes a further mismatch between the denotational and operational treatment of sums. Indeed the denotational treatment of recursive processes will lead to acyclic transition systems, which are generally not obtained with the present operational semantics. Less problematic is the fact that from the very way they are defined the transition systems obtained operationally must consist only of reachable states and transitions. This property is not preserved by the categorical operation of restriction used in the denotational semantics.

Of course, if we use a coarser relation of equivalence than isomorphism then the two semantics can be related. In the next section, it will be shown that, given any term, there is a strong bisimulation (in the sense of [Milner, 1989]) between the reachable states of the transition system obtained denotationally and those got from the operational semantics.

### 3.2 Operational semantics (version 2)

The denotational and operational approaches can be reconciled in a simple way. The idea is to modify the operational semantics, to introduce new copies of states where they are required by the denotational semantics. New copies of states are got by tagging terms by 0, 1, or 2. States for the operational semantics are built from closed terms from the syntax extended to include the clauses

$$t ::= \dots \mid (0, t) \mid (1, t) \mid (2, t).$$

We call such terms *tagged terms*—note that they include the ordinary terms.

The modified operational semantics for tagged terms is given by these rules:

$$\frac{t \xrightarrow{a} t'}{(n, t) \xrightarrow{a} (n, t')}$$

$$\overline{at \xrightarrow{a} t} \quad \overline{t \xrightarrow{*} t}$$

$$\frac{t_0 \xrightarrow{a} t'_0}{t_0 \oplus t_1 \xrightarrow{a} (0, t'_0)} a \neq * \quad \frac{t_1 \xrightarrow{b} t'_1}{t_0 \oplus t_1 \xrightarrow{b} (1, t'_1)} b \neq *$$

$$\frac{t_0 \xrightarrow{a} t'_0 \quad t_1 \xrightarrow{b} t'_1}{t_0 \times t_1 \xrightarrow{a \times b} t'_0 \times t'_1}$$

$$\frac{t \xrightarrow{a} t'}{t \upharpoonright \Lambda \xrightarrow{a} t' \upharpoonright \Lambda} a \in \Lambda \quad \frac{t \xrightarrow{a} t'}{t\{\Xi\} \xrightarrow{\Xi(a)} t'\{\Xi\}}$$

$$\frac{t[\text{rec } x.t/x] \xrightarrow{a} t'}{\text{rec } x.t \xrightarrow{a} (2, t')} a \neq *.$$

The first rule expresses that a tagged term has the capabilities of an untagged term. Notice that the former operational semantics is obtained by stripping away the tags, and in fact such a relation is a bisimulation, in the sense of Milner and Park [Milner, 1989], between the transition systems of the two forms of operational semantics.

Now we can establish a close correspondence between the operational and denotational semantics.

**Definition 3.2.1.** Letting  $T$  be the transition system of the operational semantics, with initial state a tagged term  $t$ , define

$$\mathcal{Op}(t) = \mathcal{R}(T).$$

**Lemma 3.2.2.** For any closed tagged term  $t$ , the transition system  $\mathcal{Op}(t)$  is acyclic.

**Proof.** We show this by mapping tagged terms  $t$  to  $|t|$  in a strict order  $<$  (an irreflexive, transitive relation) in such a way that

$$t \xrightarrow{a} u \ \& \ a \neq * \Rightarrow |t| < |u|. \quad (1)$$

It then follows that  $\rightarrow^+$  is irreflexive. The full proof, with the definition of  $<$ , is given in Appendix C. ■

**Theorem 3.2.3.** *Let  $t$  be a closed term of the process language **Proc**. For any environment  $\rho$*

$$\mathcal{O}p(t) \cong \mathcal{R}(\mathbf{T}[\![t]\!]\rho),$$

*a label-preserving isomorphism.*

**Proof.** By structural induction we show if  $t$  is a term with free variables  $x_1, \dots, x_k$  then for all closed terms  $t_1, \dots, t_k$ ,

$$\mathcal{O}p(t[t_1/x_1, \dots, t_k/x_k]) \cong \mathcal{R}(\mathbf{T}[\![t]\!]\rho[\mathcal{O}p(t_1)/x_1, \dots, \mathcal{O}p(t_k)/x_k]),$$

a label-preserving isomorphism. Henceforth in this proof we will use vector notation, writing, for example,  $\bar{x}$  for  $x_1, \dots, x_k$  and  $\bar{T}/\bar{x}$  for  $T_1/x_1, \dots, T_k/x_k$ .

The basis cases, when  $t$  is *nil* or a variable, hold trivially. The case where  $t$  is a prefix uses acyclicity of the operational semantics in order to ensure disjointness of the initial state, as does that of sum where, as we have seen, we require components to be nonrestarting in order for the categorical sum to reflect that given operationally. The denotational and operational semantics of the operations product, restriction, and relabelling correspond closely making the proof simple in these cases. The only case of difficulty is that where  $t$  has the form *rec y.u*.

Assume *rec y.u* has free variables  $\bar{x}$  and that  $\bar{s}$  are closed terms to instantiate  $\bar{x}$ . Writing  $v = u[s/\bar{x}]$ , we observe that from acyclicity (lemma 3.2.2) it follows that

$$\mathcal{O}p(\text{rec } y.v) \cong \mathcal{O}p(v[\text{rec } y.v/y])$$

—the isomorphism acts so

$$\begin{aligned} \text{rec } y.v &\mapsto v[\text{rec } y.v/y] \\ (2, r) &\mapsto r \end{aligned}$$

and is clearly label preserving. Hence

$$\begin{aligned} \mathcal{O}p(\text{rec } y.v) &\cong \mathcal{O}p(v[\text{rec } y.v/y]) \\ &= \mathcal{O}p(u[\bar{s}/\bar{x}, \text{rec } y.v/y]) \\ &\cong \mathcal{R}(\mathbf{T}[\![u]\!]\rho[\overline{\mathcal{O}p(s)}/\bar{x}, \mathcal{O}p(\text{rec } y.v)/y]) \end{aligned}$$

where the latter isomorphism is label preserving by the induction hypothesis. Thus  $\mathcal{O}p(\text{rec } y.v) \cong \mathcal{R}(F(\mathcal{O}p(\text{rec } y.v)))$ , also label preserving, where

$$F(T) = \mathbf{T}[\![u]\!]\rho[\overline{\mathcal{O}p(s)}/\bar{x}, T/y].$$

But  $\mathbf{T}[\text{rec } y \ u] \rho[\overline{\mathcal{O}p(s)/x}] = \text{fix}(F)$ , and so by lemma 3.0.2,

$$\mathcal{O}p(\text{rec } y. u[\overline{s/x}]) = \mathcal{O}p(\text{rec } y. v) \cong \mathcal{R}(\mathbf{T}[\text{rec } y. u] \rho[\overline{\mathcal{O}p(s)/x}]),$$

the label-preserving isomorphism required for this case of the induction. ■

There is another way in which the operational and denotational semantics agree. There is a strong bisimulation (in the sense of [Milner, 1989]) between the reachable states of the transition system obtained as the denotational semantics of a term and those got from the operational semantics. In fact, the bisimulation can be expressed as a special kind of morphism on transition systems, called *zig-zag* morphisms by van Benthem—*cf.* the chapter [Stirling, 1992].

**Definition 3.2.4.** A morphism of transition systems  $f : T \rightarrow T'$  is called a *zig-zag* morphism iff both  $T$  and  $T'$  have the same labelling set  $L$ , the morphism has the form  $f = (\sigma, 1_L)$  (*i.e.*  $f$  preserves labels), and

$$\sigma(s) \xrightarrow{a} u \text{ in } T' \Rightarrow \exists s'. s \xrightarrow{a} s' \text{ in } T'$$

for all states  $s, u$  of  $T$  and labels  $a$ .

For a closed term  $t$ , let  $T_1$  be the transition system obtained from the operational semantics (version 1) with initial state  $t$ . Write  $\mathcal{O}p_1(t)$  for  $\mathcal{R}(T_1)$ . It is easiest to relate the two forms of operational semantics by the obvious function *untag*, taking a tagged term to its associated term without tags. The isomorphism of theorem 3.2.3 then yields a zig-zag morphism relating the denotation of a closed term to its operational semantics (version 1).

**Proposition 3.2.5.** *Let  $t$  be a closed term. There is a unique morphism of transition systems*

$$\mathcal{O}p(t) \rightarrow \mathcal{O}p_1(t)$$

*which takes a state  $s$  to  $\text{untag}(s)$ ; it is a zig-zag morphism.*

**Proof.** Any derivation according to version 1 of the operational semantics is matched by one according to version 2, and *vice versa*. ■

### 3.3 An example

We will illustrate the different models on an example where the parallel composition is given by the following synchronisation algebra. Labels have the form  $a?$ ,  $a!$ ,  $a$  which intuitively can be thought of as representing receiving on channel  $a$  ( $a?$ ), sending on channel  $a$  ( $a!$ ), and completed synchronisation on channel  $a$  (simply  $a$ ). The synchronisation algebra is given by the following table.



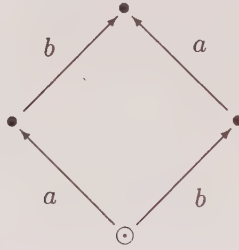


transition system does not capture concurrency in the sense that we expect that a breakdown ( $b$ ) can occur in parallel with the customer receiving coffee ( $c$ ) and this is not caught by the transition system. This limitation of transition systems can be seen even more starkly for the two simple terms

$$a\ b\ nil \oplus b\ a\ nil$$

$$a\ nil \parallel b\ nil$$

both of which can be described by the same transition system, *viz.*



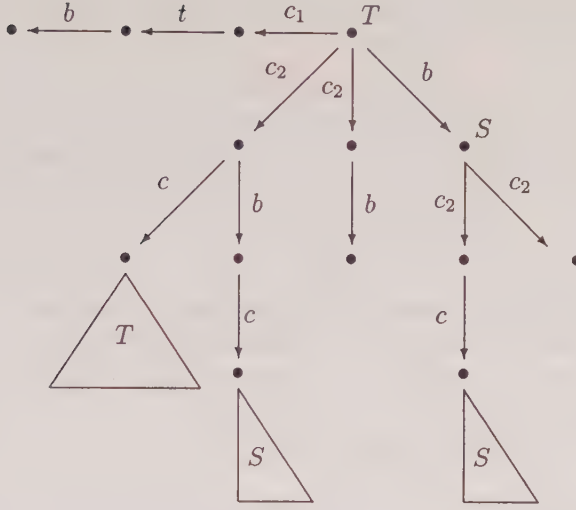
This contrasts the interleaving model of transition systems with noninterleaving models, like the Petri nets we shall see later, which represent independence of actions explicitly.

## 4 Synchronisation trees

We now turn to consider another model. It gives rise to our first example illustrating how different models can be related through the help of adjunctions between their associated categories.

In his foundational work on CCS [Milner, 1980], Milner introduced synchronisation trees as a model of parallel processes and explained the meaning of the language of CCS in terms of operations on them. In this section we briefly examine the category of synchronisation trees and its relation to that of labelled transition systems. This illustrates the method by which many other models are related, and the role category theoretic ideas play in formulating and proving facts which relate semantics in one model to semantics in another.

**Example 4.0.1.** We return to the example of 3.3. As a synchronisation tree  $SYS$  could be represented by



where the trees stemming from  $S$  and  $T$  are repeated as indicated. The synchronisation tree is obtained by unfolding the transition system of 3.3. Such an unfolding operation arises as an adjoint in the formulation of the models as categories. In moving to synchronisation trees we have lost the cyclic structure of the original transition system: that the computation can repeatedly visit the same state. We can still detect the possibility of deadlock if the customer inserts coin  $c_2$ .

As we have seen, a synchronisation tree is a tree together with labels on its arcs. Formally, we define synchronisation trees to be special kinds of labelled transition systems: those for which the transition relation is acyclic and can only branch away from the root.

**Definition 4.0.2.** A *synchronisation tree* is a transition system  $(S, i, L, \text{Tran})$  where

- every state is reachable,
- if  $s \xrightarrow{v} s$ , for a string  $v$ , then  $v$  is empty (i.e. the transition system is acyclic), and
- $s' \xrightarrow{a} s \ \& \ s'' \xrightarrow{b} s \Rightarrow a = b \ \& \ s' = s''$ .

Regarded in this way, we obtain synchronisation trees as a full subcategory of labelled transition systems, with a projection functor to the category of labelling sets with partial functions.

**Definition 4.0.3.** Write  $\mathbf{S}$  for the full subcategory of synchronisation trees in  $\mathbf{T}$ .

In fact, the inclusion functor  $\mathbf{S} \hookrightarrow \mathbf{T}$  has a right adjoint  $ts : \mathbf{T} \rightarrow \mathbf{S}$  which has the effect of unfolding a labelled transition system to a synchronisation tree.<sup>3</sup>

**Definition 4.0.4.** Let  $T$  be a labelled transition system  $(S, i, L, Tran)$ . Define  $ts(T)$  to be  $(S', i', L, Tran')$  where:

- The set  $S'$  consists of all finite, possibly empty, sequences of transitions

$$(t_1, \dots, t_j, t_{j+1}, \dots, t_{n-1})$$

such that  $t_j = (s_{j-1}, a_j, s_j)$  and  $t_{j+1} = (s_j, a_{j+1}, s_{j+1})$  whenever  $1 < j < n$ . The element  $i' = ()$ , the empty sequence.

- The set  $Tran'$  consists of all triples  $(u, a, v)$  where  $u, v \in S'$  and  $u = (u_1, \dots, u_k), v = (u_1, \dots, u_k, (s, a, s'))$ , obtained by appending an  $a$  transition to  $u$ .

Define  $\phi : S' \rightarrow S$  by taking  $\phi(()) = i$  and  $\phi((t_1, \dots, t_n)) = s_n$ , where  $t_n = (s_{n-1}, a_n, s_n)$ .

**Theorem 4.0.5.** Let  $T$  be a labelled transition system, with labelling set  $L$ . Then  $ts(T)$  is a synchronisation tree, also with labelling set  $L$ , and, with the definition above,  $(\phi, 1_L) : ts(T) \rightarrow T$  is a morphism. Moreover,  $ts(T), (\phi, 1_L)$  is cofree over  $T$  with respect to the inclusion functor  $\mathbf{S} \hookrightarrow \mathbf{T}$ , i.e. for any morphism  $f : V \rightarrow T$ , with  $V$  a synchronisation tree, there is a unique morphism  $g : V \rightarrow ts(T)$  such that  $f = (\phi, 1_L) \circ g$ :

$$\begin{array}{ccc} T & \xleftarrow{(\phi, 1_L)} & ts(T) \\ & \nwarrow f & \uparrow g \\ & & V \end{array}$$

**Proof.** Let  $T$  be a labelled transition system, with labelling set  $L$ . It is easily seen that  $ts(T)$  is a labelled transition system with labelling set  $L$  and  $(\phi, 1_L) : ts(T) \rightarrow T$  is a morphism. To show the cofreeness property, let  $f = (\sigma, \lambda) : V \rightarrow T$  be a morphism from a synchronisation tree  $V$ . We require the existence of a unique morphism  $g : V \rightarrow ts(T)$  such that  $f = (\phi, 1_L)g$ . The morphism  $g$  must necessarily have the form  $g = (\sigma_1, \lambda)$ . The map  $\sigma_1$  is defined by induction on the distance from the root of states of  $V$ , as follows:

On the initial state  $i_V$  of  $V$ , we take  $\sigma_1(i_V) = ()$ . For any state  $v'$  for which  $(v, a, v')$  is a transition of  $V$  we take  $\sigma_1(v') = \sigma_1(v)$  if  $\lambda(a) = *$ ; otherwise, in the case where  $\lambda(a)$  is defined, we take  $\sigma_1(v') = \sigma_1(v)((\sigma(v), \lambda(a), \sigma(v')))$ .

<sup>3</sup>Because we shall be concerned with several categories and functors between them we name the functors in a way that indicates their domain and range.

It follows by induction on the distance of states  $v$  from the root that  $\sigma(v) = \phi\sigma_1(v)$ , and that  $(\sigma_1, \lambda)$  is the unique morphism such that  $f = (\phi, 1_L)g$ . (For a very similar, but more detailed, argument see [Winskel, 1985].) ■

It follows that the operation  $ts$  extends to a functor which is right adjoint to the inclusion functor from  $\mathbf{S}$  to  $\mathbf{T}$  and that the morphisms  $(\phi, 1_L) : ts(T) \rightarrow T$  are the counits of this adjunction (see [MacLane, 1971] theorem 2, p.81). This makes  $\mathbf{S}$  a (full) coreflective subcategory of  $\mathbf{T}$ , which implies the intuitively obvious fact that a synchronisation tree  $T$  is isomorphic to its unfolding  $ts(T)$  (see [MacLane, 1971] p.88).

Like transition systems, synchronisation trees have been used to give semantics to languages like CCS and CSP (see *e.g.* [Milner, 1980; Brookes, 1983]). Nondeterministic sums of processes are modelled by the operation of joining synchronisation trees at their roots, a special case of the nondeterministic sum of transition systems. We use  $\sum_{i \in I} S_i$  for the sum of synchronisation trees indexed by  $i \in I$ . For the semantics of parallel composition, use is generally made of Milner's "expansion theorem" (see [Milner, 1980]). In our context, the expansion of a parallel composition as a nondeterministic sum appears as a characterisation of the product of synchronisation trees.

**Proposition 4.0.6.** *The product of two synchronisation trees  $S$  and  $T$  of the form*

$$S \cong \sum_{i \in I} a_i S_i \quad \text{and} \quad T \cong \sum_{j \in J} b_j T_j$$

*is given by*

$$S \times T \cong \sum_{i \in I} (a_i, *) S_i \times T \oplus \sum_{i \in I, j \in J} (a_i, b_j) S_i \times T_j \oplus \sum_{j \in J} (*, b_j) S \times T_j.$$

**Proof.** The fact that the category of synchronisation trees has products and that they are preserved by the unfolding operation  $ts$  is a consequence of the general fact that right adjoints preserve limits. In particular,  $ts(S \times_T T)$  is a product of the synchronisation trees  $S$  and  $T$  above; the proof that products of trees have the form claimed follows by considering the sequences of executable transitions of  $S \times_T T$ . ■

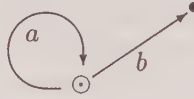
The coreflection between transition systems and synchronisation trees is fibrewise in that it restricts to adjunctions between fibres over a common labelling set. For example, for this reason its right adjoint of unfolding automatically preserves restriction (see Appendix B). In fact, via the coreflection  $\mathbf{S}$  inherits a bifibration structure from  $\mathbf{T}$ . As the following example shows, right adjoints, such as the operation of unfolding a transition system to a tree, do not necessarily preserve colimits like nondeterministic sums.



**Example 4.0.7.** Recall that the fibred coproduct  $T_0 +_L T_1$  of



both assumed to have the labelling set  $L = \{a, b\}$ , is



It is easily seen that  $ts(T_0 +_L T_1)$  is not isomorphic to  $ts(T_0) +_L ts(T_1)$ .

## 5 Languages

Synchronisation trees abstract away from the looping structure of processes. Now we examine a yet more abstract model: that of languages which further ignore the nondeterministic branching structure of processes.

**Definition 5.0.1.** A *language* over a labelling set  $L$  consists of  $(H, L)$  where  $H$  is a nonempty subset of strings  $L^*$  which is closed under prefixes, i.e. if  $a_0 \cdots a_{i-1} a_i \in H$  then  $a_0 \cdots a_{i-1} \in H$ .

Thus for a language  $(H, L)$  the empty string  $\varepsilon$  is always contained in  $H$ . Such languages were called *traces* in [Hoare, 1981] and for this reason, in the context of modelling concurrency, they are sometimes called *Hoare traces*. They consist, however, simply of strings and are not to be confused with the traces of Mazurkiewicz, to be seen later.

**Example 5.0.2.** Refer back to the customer-vending machine example of 3.3. The semantics of *SYS* as a language (its Hoare traces), loses the nondeterministic structure present in both the transition-system and synchronisation-tree descriptions. The language determined by *SYS* is

$$\{\varepsilon, c_1, c_2, b, c_1t, c_2c, c_2b, bc_2, \dots\}.$$

Lost is the distinction between, for instance, the two branches of computation  $c_2b$ , one which can be resumed by further computation and the other which deadlocks.

Morphisms of languages are partial functions on their alphabets which send strings in one language to strings in another:

**Definition 5.0.3.** A partial function  $\lambda : L \rightarrow_* L'$  extends to strings by defining

$$\widehat{\lambda}(sa) = \begin{cases} \widehat{\lambda}(s)\lambda(a) & \text{if } \lambda(a) \text{ defined,} \\ \widehat{\lambda}(s) & \text{if } \lambda(a) \text{ undefined.} \end{cases}$$

A *morphism* of languages  $(H, L) \rightarrow (H', L')$  consists of a partial function  $\lambda : L \rightarrow_* L'$  such that  $\forall s \in H. \widehat{\lambda}(s) \in H'$ .

We write  $\mathbf{L}$  for the category of languages with the above understanding of morphisms, where composition is our usual composition of partial functions.

Ordering strings in a language by extension enables us to regard the language as a synchronisation tree. The ensuing notion of morphism coincides with that of languages. This observation yields a functor from  $\mathbf{L}$  to  $\mathbf{S}$ . On the other hand any transition system, and in particular any synchronisation tree, gives rise to a language consisting of strings of labels obtained from the sequences of transitions it can perform. This operation extends to a functor. The two functors form an adjunction from  $\mathbf{S}$  to  $\mathbf{L}$  (but not from  $\mathbf{T}$  to  $\mathbf{L}$ ).

**Definition 5.0.4.** Let  $(H, L)$  be a language. Define  $ls(H, L)$  to be the synchronisation tree  $(H, \varepsilon, L, tran)$  where

$$(h, a, h') \in Tran \Leftrightarrow h' = ha.$$

Let  $T = (S, i, L, Tran)$  be a synchronisation tree. Define  $sl(T) = (H, L)$  where a string  $h \in L^*$  is in the language  $H$  iff there is a sequence, possibly empty, of transitions

$$i \xrightarrow{a_1} s_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} s_n$$

in  $T$  such that  $h = a_1 a_2 \dots a_n$ . Extend  $sl$  to a functor by defining  $sl(\sigma, \lambda) = \lambda$  for  $(\sigma, \lambda) : T \rightarrow T'$  a morphism between synchronisation trees.

**Theorem 5.0.5.** Let  $(H, L)$  be a language. Then  $ls(H, L)$  is a synchronisation tree, with labelling set  $L$ , and  $1_L : sl \circ ls(H, L) \rightarrow (H, L)$  is an isomorphism. Moreover,  $sl \circ ls(H, L), 1_L$  is cofree over  $(H, L)$  with respect to the functor  $sl : \mathbf{S} \rightarrow \mathbf{L}$ , i.e. for any morphism  $\lambda : sl(T) \rightarrow (H, L)$ , with  $T$  a synchronisation tree, there is a unique morphism  $g : T \rightarrow ls(H, L)$  such that  $\lambda = 1_L \circ sl(g)$ :

$$\begin{array}{ccc} (H, L) & \xleftarrow{1_L} & sl \circ ls(H, L) \\ & \nwarrow \lambda & \uparrow sl(g) \\ & & sl(T) \end{array}$$

**Proof.** Each state  $s$  of the synchronisation tree  $T$  is associated with a unique sequence of transitions

$$i \xrightarrow{a_1} s_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} s_n$$

with  $s_n = s$ . Defining  $\sigma(s)$  to be  $\widehat{\lambda}(a_1 \dots a_n)$  makes  $(\sigma, \lambda)$  the unique morphism  $g : T \rightarrow ls(H, L)$  such that  $\lambda = 1_L \circ sl(g)$ .  $\blacksquare$

This demonstrates the adjunction  $\mathbf{S} \rightarrow \mathbf{L}$  with left adjoint  $sl$  and right adjoint  $ls$ ; the fact that the counit is an isomorphism makes the adjunction a (full) reflection. Let  $r : \mathbf{L} \rightarrow \mathbf{Set}_*$  be the functor sending a morphism  $\lambda : (H, L) \rightarrow (H', L')$  of languages to  $\lambda : L \rightarrow_* L'$ . Let  $q : \mathbf{S} \rightarrow \mathbf{Set}_*$  be the functor sending synchronisation trees to their labelling sets (a restriction of the functor  $p$  from transition systems). With respect to these projections the adjunction is fibred.

We can immediately observe some categorical constructions. The fibre product and coproduct are simply the intersection and union of languages over the same labelling set. The product of two languages  $(H_0, L_0), (H_1, L_1)$  takes the form

$$(\widehat{\pi}_0^{-1} H_0 \cap \widehat{\pi}_1^{-1} H_1, L_0 \times_* L_1),$$

with projections  $\pi_0 : L_0 \times_* L_1 \rightarrow_* L_0$  and  $\pi_1 : L_0 \times_* L_1 \rightarrow_* L_1$  obtained from the product in  $\mathbf{Set}_*$ . The coproduct of languages  $(H_0, L_0), (H_1, L_1)$  is

$$(\widehat{j}_0 H_0 \cup \widehat{j}_1 H_1, L_0 \uplus L_1)$$

with injections  $j_0 : L_0 \rightarrow L_0 \uplus L_1, j_1 : L_1 \rightarrow L_0 \uplus L_1$  into the left and right component of the disjoint union. The fibre product and coproduct of languages over the same alphabet are given simply by intersection and union respectively.

The expected constructions of restriction and relabelling arise as (strong) cartesian and cocartesian liftings.

## 6 Relating semantics

We can summarise the relationship between the different models by recalling the coreflection and reflection (and introducing a little notation to depict such adjunctions):

$$\mathbf{L} \longleftarrow \triangleleft \mathbf{S} \longrightarrow \triangleright \mathbf{T}$$

The coreflection and reflection are associated with “inclusions”, embedding one model in another—the direction of the embedding being indicated by the hooks on the arrows, whose tips point in the direction of the left adjoint. Each inclusion has an adjoint; the inclusion of the coreflection has a right and that of the reflection a left adjoint. These functors from right to left

correspond respectively to losing information about the looping, and then in addition the nondeterministic branching structure of processes.<sup>4</sup>

Such categorical facts are useful in several ways. The coreflection  $\mathbf{S} \dashv \mathbf{T}$  tells us how to construct limits in  $\mathbf{S}$  from those in  $\mathbf{T}$ . In particular, we have seen how the form of products in  $\mathbf{S}$  is determined by their simpler form in  $\mathbf{T}$ . We regard synchronisation trees as transition systems via the inclusion functor, form the limit there, and then transport it to  $\mathbf{S}$ , using the fact that right adjoints preserve limits. Because the adjunctions are fibrewise, the right adjoints also preserve cartesian liftings and left adjoints cocartesian liftings (as is shown in Appendix B, lemma B.0.5). The fact that the embedding functors are full and faithful ensures that they reflect limits and colimits, as well as cartesian and cocartesian morphisms because the adjunctions are fibrewise (lemma B.0.6).

Imagine giving semantics to the process language **Proc** of section 3 in any of the three models. Any particular construct is interpreted as being built up in the same way from universal constructions. For example, product in the process language is interpreted as categorical product, and nondeterministic sum in the language as the same combination of cocartesian liftings and coproduct we use in transition systems. Constructions are interpreted in a uniform manner in any of the different models. Prefixing for languages requires a (straightforward) definition. Recursion requires a separate treatment. Synchronisation trees can be ordered in the same way as transition systems. Languages can be ordered by inclusion. In both cases it is straightforward to give a semantics. With respect to an environment  $\rho_S$  from process variables to synchronisation trees we obtain a denotational semantics yielding a synchronisation tree

$$\mathbf{S}[t]_{\rho_S}$$

for any process term  $t$ . And with respect to an environment  $\rho_L$  from process variables to languages the denotational semantics yields a language

$$\mathbf{L}[t]_{\rho_L}$$

for a process term  $t$ . What is the relationship between the three semantics

$$\mathbf{T}[-], \mathbf{S}[-], \text{ and } \mathbf{L}[-]?$$

Consider the relationship between the semantics in transition systems and synchronisation trees. Letting  $\rho$  be an environment from process variables,

---

<sup>4</sup>**Warning:** We use the term “coreflection” to mean an adjunction in which the unit is a natural isomorphism, or equivalently (by theorem 1, p.89 of [MacLane, 1971]) when the left adjoint is full and faithful. Similarly, “reflection” is used here to mean an adjunction for which the counit is a natural isomorphism, or equivalently when the right adjoint is full and faithful. While the same uses can be found in the literature, they are not entirely standard.

to transition systems, the two semantics are related by

$$ts(\mathbf{T}[t]\rho) = \mathbf{S}[t]ts \circ \rho$$

for any process term  $t$ . This is proved by structural induction on  $t$ . The cases where  $t$  is a product or restriction follow directly from preservation properties of right adjoints. The other cases require special, if easy, argument. For example, the fact that

$$ts(\mathbf{T}[t_0 \oplus t_1]\rho) = \mathbf{S}[t_0 \oplus t_1]ts \circ \rho$$

depends on  $\mathbf{T}[t_0]\rho, \mathbf{T}[t_1]\rho$  being nonrestarting, a consequence of acyclicity (lemma 3.2.2) shown earlier. The case of recursion requires the  $\trianglelefteq$ -continuity of the unfolding functor  $ts$ . A similar relationship,

$$sl(\mathbf{S}[t]\rho) = \mathbf{L}[t]sl \circ \rho$$

for a process term  $t$ , and environment  $\rho$  to synchronisation trees, holds between the two semantics in synchronisation trees and languages. This time the structural induction is most straightforward in the cases of *nil*, nondeterministic sum, and relabelling (because of the preservation properties of the left adjoint  $sl$ ). However, simple arguments suffice for the other cases.

In summary, the operations on processes are interpreted in a uniform manner (with the same universal constructions) in the three different semantics. The preservation properties of adjoints are useful in relating the semantics. Less directly, a knowledge of what we can and cannot expect to be preserved automatically provides useful guidelines in itself. The failure of a general preservation property can warn that the semantics of a construct can only be preserved in special circumstances. For instance, we cannot expect a right adjoint like  $ts$  always to preserve a colimit, like a nondeterministic sum. Accordingly, the semantics of sums is only preserved by  $ts$  by virtue of a special circumstance: that the transition systems denoted are nonrestarting. The advantages of a categorical approach become more striking when we turn to the more intricate models of the noninterleaving approach to concurrency, but where again the same universal constructions will be used.

## 7 Trace languages

All the models we have considered so far have identified concurrency, or parallelism, with nondeterministic interleaving of atomic actions. We turn now to consider models where concurrency is modelled explicitly in the form of independence between actions. In some models, like Mazurkiewicz traces, the relation of independence is a basic notion while in others, like Petri nets,



it is derived from something more primitive. The idea is that if two actions are enabled and also independent then they can occur concurrently, or in parallel. Models of this kind are sometimes said to capture “true concurrency”, a convenient though regrettably biased expression. They are also often called “noninterleaving models” though this again is inappropriate; as we shall see, Petri nets can be described as forms of transition systems. A much better term is “independence models” for concurrent computation, though this is not established. Because in such models the independence of actions is not generally derivable from an underlying property of their labels, depending rather on which occurrences are considered, we will see an important distinction basic to these richer models. They each have a concept of *events* distinguished from that of *labels*. Events are to be thought of as atomic actions which can support a relation of independence. Events can then bear the further structure of having a label, for instance signifying which channel or which process they belong to.

A greater part of the development of these models is indifferent to the extra labelling structure we might like to impose, though of course restriction and relabelling will depend on labels. Our treatment of the models and their relationship will be done primarily for the unlabelled structures. Later we will adjoin labelling and provide semantics in terms of the various models and discuss their relationship.

## 7.1 A category of trace languages

The simplest model of computation with an in-built notion of independence is that of Mazurkiewicz trace languages. They are languages in which the alphabet also possesses a relation of independence. As we shall see, this small addition has a striking effect in terms of the richness of the associated structures. It is noteworthy that, in applications of trace languages, there have been different understandings of the alphabet; in Mazurkiewicz’s original work the alphabet is thought of as consisting of events (especially events of a Petri net), while some authors have instead interpreted its elements as labels, for example standing for port names. This remark will be elaborated later on in section 7.2.

**Definition 7.1.1.** A *Mazurkiewicz trace language* consists of  $(M, L, I)$  where  $L$  is a set,  $I \subseteq L \times L$  is a symmetric, irreflexive relation called the *independence* relation, and  $M$  is a nonempty subset of strings  $L^*$  such that

- *prefix closed*:  $sa \in M \Rightarrow s \in M$  for all  $s \in L^*, a \in L$ ,
- *I-closed*:  $sab \in M \ \& \ aIb \Rightarrow sbat \in M$  for all  $s, t \in L^*, a, b \in L$ ,
- *coherent*:  $sa \in M \ \& \ sb \in M \ \& \ aIb \Rightarrow sab \in M$  for all  $s \in L^*, a, b \in L$ .

The alphabet  $L$  of a trace language  $(M, L, I)$  can be thought of as the set of actions of a process and the set of strings as the sequences of actions the process can perform. Some actions are independent of others. The axiom of  $I$ -closedness expresses a consequence of independence: if two actions are independent and can occur one after the other then they can occur in the opposite order. The axiom of coherence is not generally imposed. We find it convenient (though not essential for a great deal that follows), and besides, like  $I$ -closedness, it seems to follow from an intuitive understanding of what independence means; it says that if two actions are independent and both can occur from the same state then they can occur one after the other, in either order. Given that some actions are independent of others, it is to be expected that some strings represent essentially the same computation as others. For example, if  $a$  and  $b$  are independent then both strings  $ab$  and  $ba$  represent the computation of  $a$  and  $b$  occurring concurrently. More generally, two strings  $sabt$  and  $sbat$  represent the same computation when  $a$  and  $b$  are independent. This extends to an equivalence relation between strings, the equivalence classes of which are called *Mazurkiewicz traces*. There is an associated preorder between strings of a trace language which induces a partial order on traces.

**Definition 7.1.2.** Let  $(M, L, I)$  be a trace language. For  $s, t \in M$  define  $\approx$  to be the smallest equivalence relation such that

$$sabt \approx sbat \text{ if } aIb$$

for  $sabt, sbat \in M$ . Call an equivalence class  $\{s\}_\approx$ , for  $s \in M$ , a *trace*. For  $s, t \in M$  define

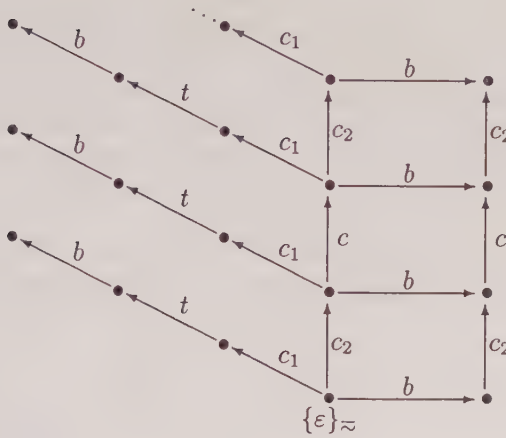
$$s \lesssim t \Leftrightarrow \exists u. su \approx t.$$

**Proposition 7.1.3.** Let  $(M, L, I)$  be a trace language with trace equivalence  $\approx$ . If  $su \in M$  and  $s \approx s'$  then  $s'u \in M$  and  $su \approx s'u$ . The relation  $\lesssim$  of the trace language is a preorder. Its quotient  $\lesssim / \approx$  by the equivalence relation  $\approx$  is a partial order on traces.

**Example 7.1.4.** The independence relation of Mazurkiewicz allows us to express the concurrency we remarked on earlier in example 3.3. By asserting that the independence relation  $I$  is the smallest such that

$$bIc \quad \text{and} \quad bIc_2,$$

corresponding to the idea that a breakdown  $b$  can occur in parallel with the customer receiving coffee, the language of example 5.0.2 collapses under the identifications of trace equivalence to give the following ordering  $\lesssim / \approx$  on traces:



We have drawn the traces as points and drawn an arrow  $\xrightarrow{a}$  from a trace  $\{s\}_{\approx}$  to a trace  $\{sa\}_{\approx}$ . Although the potential concurrency of  $b$  and  $c$ , and  $b$  and  $c_2$ , is caught by the independence relation, this trace-language semantics, like the language semantics before, is blind to the fact that the system can deadlock after the customer inserts a coin  $c_2$ . To make such a distinction we would have to distinguish the two kinds of occurrences of  $c_2$ , regarding them as different events.

The partial order  $\lesssim / \approx$  of a trace language can be associated with a partial order of causal dependencies between event occurrences. This structure will be investigated in the next section.

Morphisms between trace languages are morphisms between the underlying languages which preserve independence:

**Definition 7.1.5.** A morphism of trace languages  $(M, L, I) \rightarrow (M', L', I')$  consists of a partial function  $\lambda : L \rightarrow_* L'$  which

- preserves independence:  $aIb$  &  $\lambda(a)$  defined &  $\lambda(b)$  defined  $\Rightarrow \lambda(a)I'\lambda(b)$  for all  $a, b \in L$ ,
- preserves strings:  $s \in M \Rightarrow \hat{\lambda}(s) \in M'$  for all strings  $s$ .

It is easy to see that morphisms of trace languages preserve traces and the ordering between them:

**Proposition 7.1.6.** Let  $\lambda : (M, L, I) \rightarrow (M', L', I')$  be a morphism of trace languages. If  $s \lesssim t$  in the trace language  $(M, L, I)$  then  $\hat{\lambda}(s) \lesssim \hat{\lambda}(t)$  in the trace language  $(M', L', I')$ .

**Definition 7.1.7.** Write **TL** for the category of trace languages with composition that of partial functions.

## 7.2 Constructions on trace languages

We examine some categorical constructions on trace languages. The constructions generalise from those on languages but with the added consideration of defining independence.

Let  $(M_0, L_0, I_0)$  and  $(M_1, L_1, I_1)$  be trace languages. Their *product* is  $(M, L, I)$  where  $L = L_0 \times_* L_1$ , the product in  $\mathbf{Set}_*$ , with projections  $\pi_0 : L \rightarrow_* L_0$  and  $\pi_1 : L \rightarrow_* L_1$ , with

$$\begin{aligned} aIb \Leftrightarrow & (\pi_0(a), \pi_0(b) \text{ defined} \Rightarrow \pi_0(a)I_0\pi_0(b)) \ \& \\ & (\pi_1(a), \pi_1(b) \text{ defined} \Rightarrow \pi_1(a)I_1\pi_1(b)), \end{aligned}$$

and

$$M = \widehat{\pi}_0^{-1}M_0 \cap \widehat{\pi}_1^{-1}M_1.$$

Their *coproduct* is  $(M, L, I)$  where  $L = L_0 \uplus L_1$ , the disjoint union, with injections  $j_0 : L_0 \rightarrow L$ ,  $j_1 : L_1 \rightarrow L$ ; the relation  $I$  satisfies

$$\begin{aligned} aIb \Leftrightarrow & \exists a_0, b_0. a_0I_0b_0 \ \& \ a = j_0(a_0) \ \& \ b = j_0(b_0) \ \text{or} \\ & \exists a_1, b_1. a_1I_1b_1 \ \& \ a = j_1(a_1) \ \& \ b = j_1(b_1) \end{aligned}$$

and

$$M = \widehat{j}_0M_0 \cup \widehat{j}_1M_1.$$

What about restriction and relabelling? Restriction appears again as a cartesian lifting of an inclusion between labelling sets. Its effect is simply to cut down the language and independence to the restricting set. However, the relabelling of a trace language cannot always be associated with a cocartesian lifting. To see this consider a function  $\lambda : \{a, b\} \rightarrow \{\gamma\}$  sending both  $a, b$  to  $\gamma$ . If a trace language  $T$  has  $\{a, b\}$  as an alphabet and has  $a, b$  independent, then  $\lambda$  cannot be a morphism of trace languages, and hence no cocartesian lifting of  $\lambda$  with respect to the trace language  $T$ ; because independence is irreflexive, independence cannot be preserved by  $\lambda$ .<sup>5</sup> The difficulty stems from our implicitly regarding the alphabet of a Mazurkiewicz trace language as a set of labels of the kind used in the operations of restriction and relabelling. Although the alphabet can be taken to have this nature, it was not the original intention of Mazurkiewicz. Here it is appropriate to discuss the two ways in which trace languages are used to model parallel processes.

One way is to use trace languages in the same manner as languages. This was implicitly assumed in our attempts to define the relabelling of a trace language, and in the example above. Then a process, for example in CCS, denotes a trace language, with alphabet the labels of the process.

---

<sup>5</sup>If, however, we instead project to the category of *sets with independence*  $\mathbf{Set}_I$  we obtain a fibration and cofibration—see section 8.3.3, in particular proposition 8.3.20.



This regards symbols of the alphabet of a trace language as labels in a process algebra. As we have seen in the treatment of interleaving models labels can be understood rather generally; they are simply tags to distinguish some actions from others. However, this general understanding of the alphabet conflicts with this first approach. As the independence relation is then one between labels, once it is decided that, say,  $a$  and  $b$  are independent in the denotation of a process then they are so throughout its execution. However, it is easy to imagine a process where at some stage  $a$  and  $b$  occur independently and yet not at some other stage. To remedy this some have suggested that the independence relation be made to depend on the trace of labels which has occurred previously. But even with this modification, the irreflexivity of the independence relation means there cannot be independent occurrences with the same label; in modelling a CCS process all its internal  $\tau$  events would be dependent!

The other approach is to regard the alphabet as consisting, not of labels of the general kind we have met in process algebras, but instead as consisting of *events*. It is the events which possess an independence relation and any distinctions that one wishes to make between them are then caught through an extra labelling function from events to labels. True, this extra level of labelling complicates the model, but the distinction between events and the labels they can carry appears to be fundamental. It is present in the other models which capture concurrency directly as independence. This second view fits with that of Mazurkiewicz's trace-language semantics of Petri nets. When we come to adjoin the extra structure of labels, restriction will again be associated with a cartesian lifting and relabelling will reappear as a cocartesian lifting.

There remains the question of understanding the order  $\lesssim/\approx$  of trace languages. We shall do this through a representation theorem which will show that  $\lesssim/\approx$  can be understood as the subset ordering between configurations of an event structure.

## 8 Event structures

There is most often no point in analysing the precise places and times of events in a distributed computation. What is generally important are the significant events and how the occurrence of an event causally depends on the previous occurrence of others. For example, the event of a process transmitting a message would presumably depend on it first performing some events, so it was in the right state to transmit, including the receipt of the message which in turn would depend on its previous transmission by another process. Such ideas suggest that we view distributed computations as event occurrences together with a relation expressing causal dependency, and this we may reasonably take to be a partial order. One thing missing from such descriptions is the phenomenon of nondeterminism. To model



nondeterminism we adjoin further structure in the form of a conflict relation between events to express how the occurrence of certain events rules out the occurrence of others. Here we shall assume that events exclude each other in a binary fashion, though variants of this have been treated.

**Definition 8.0.1.** Define an *event structure* to be a structure  $(E, \leq, \#)$  consisting of a set  $E$  of *events* which are partially ordered by  $\leq$ , the *causal dependency relation*, and a binary, symmetric, irreflexive relation  $\# \subseteq E \times E$ , the *conflict relation*, which satisfy

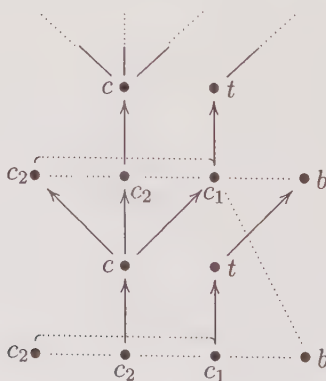
$$\begin{aligned} \{e' \mid e' \leq e\} \text{ is finite,} \\ e \# e' \leq e'' \Rightarrow e \# e'' \end{aligned}$$

for all  $e, e', e'' \in E$ .

Say two events  $e, e' \in E$  are *concurrent*, and write  $e \text{ co } e'$ , iff  $\neg(e \leq e' \text{ or } e' \leq e \text{ or } e \# e')$ . Write  $\mathbb{W}$  for  $\# \cup 1_E$ , i.e. the reflexive closure of the conflict relation.

The finiteness assumption restricts attention to discrete processes where an event occurrence depends only on finitely many previous occurrences. The axiom on the conflict relation expresses that if two events causally depend on events in conflict then they too are in conflict. Note that events of an event structure correspond to event *occurrences*.

**Example 8.0.2.** As an illustration, we examine how to represent the process  $SYS$  of 3.3 as an event structure. Strictly speaking  $SYS$  is represented as the *labelled* event structure, drawn below, where the event occurrences (the events of the event structure) appear as “•”, labelled to indicate their nature. The sequential nature of the components  $VM$ ,  $VM'$ , and  $C$  imposes a partial order of causal dependency  $\leq$ , the immediate steps of which are drawn as upwards arrows, and nondeterminism shows up as conflict, generated by the relation indicated by dotted lines.



In fact this labelled event structure is that obtained from the denotational semantics following the general scheme of section 11.2.

Guided by our interpretation we can formulate a notion of computation state of an event structure  $(E, \leq, \#)$ . Taking a computation state of a process to be represented by the set  $x$  of events which have occurred in the computation, we expect that

$$e' \in x \ \& \ e \leq e' \Rightarrow e \in x$$

—if an event has occurred then all events on which it causally depends have occurred too—and also that

$$\forall e, e' \in x. \neg(e \# e')$$

—no two conflicting events can occur together in the same computation.

**Definition 8.0.3.** Let  $(E, \leq, \#)$  be an event structure. Define its *configurations*,  $\mathcal{D}(E, \leq, \#)$ , to consist of those subsets  $x \subseteq E$  which are

- *conflict free*:  $\forall e, e' \in x. \neg(e \# e')$  and
- *downwards closed*:  $\forall e, e'. e' \leq e \in x \Rightarrow e' \in x$ .

In particular, define  $[e] = \{e' \in E \mid e' \leq e\}$ . (Note that  $[e]$  is a configuration as it is conflict free.)

Write  $\mathcal{D}^0(E, \leq, \#)$  for the set of finite configurations.

The important relations associated with an event structure can be recovered from its finite configurations (or indeed similarly from its configurations):

**Proposition 8.0.4.** Let  $(E, \leq, \#)$  be an event structure. Then

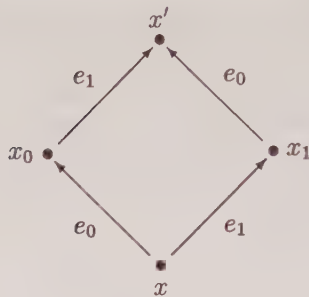
- $e \leq e' \Leftrightarrow \forall x \in \mathcal{D}^0(E, \leq, \#). e' \in x \Rightarrow e \in x$ .
- $e \# e' \Leftrightarrow \forall x \in \mathcal{D}^0(E, \leq, \#). e \in x \Rightarrow e' \notin x$ .
- $e \text{ co } e' \Leftrightarrow \exists x, x' \in \mathcal{D}^0(E, \leq, \#). e \in x \ \& \ e' \notin x \ \& \ e' \in x' \ \& \ e \notin x' \ \& \ x \cup x' \in \mathcal{D}^0(E, \leq, \#)$ .

Events manifest themselves as atomic jumps from one configuration to another, and later it will follow that we can regard such jumps as transitions in an asynchronous transition system.

**Definition 8.0.5.** Let  $(E, \leq, \#)$  be an event structure. Let  $x, x'$  be configurations. Write

$$x \xrightarrow{e} x' \Leftrightarrow e \notin x \ \& \ x' = x \cup \{e\}.$$

**Proposition 8.0.6.** Two events  $e_0, e_1$  of an event structure are in the concurrency relation *co* iff there exist configurations  $x, x_0, x_1, x'$  such that:



## 8.1 A category of event structures

We define morphisms on event structures as follows:

**Definition 8.1.1.** Let  $ES = (E, \leq, \#)$  and  $ES' = (E', \leq', \#')$  be event structures. A *morphism* from  $ES$  to  $ES'$  consists of a partial function  $\eta : E \rightarrow_* E'$  on events which satisfies

$$\begin{aligned} x \in \mathcal{D}(ES) &\Rightarrow \eta x \in \mathcal{D}(ES') \text{ \& } \\ \forall e_0, e_1 \in x. \eta(e_0), \eta(e_1) \text{ both defined} &\text{ \& } \eta(e_0) = \eta(e_1) \Rightarrow e_0 = e_1. \end{aligned}$$

A morphism  $\eta : ES \rightarrow ES'$  between event structures expresses how behaviour in  $ES$  determines behaviour in  $ES'$ . The partial function  $\eta$  expresses how the occurrence of an event in  $ES$  implies the simultaneous occurrence of an event in  $ES'$ ; the fact that  $\eta(e) = e'$  can be understood as expressing that the event  $e'$  is a “component” of the event  $e$  and, in this sense, that the occurrence of  $e$  implies the simultaneous occurrence of  $e'$ . If two distinct events in  $ES$  have the same image in  $ES'$  under  $\eta$  then they cannot belong to the same configuration.

Morphisms of event structures preserve the concurrency relation. This is a simple consequence of proposition 8.0.6, showing how the concurrency relation holding between events appears as a “little square” of configurations.

**Proposition 8.1.2.** Let  $ES$  be an event structure with concurrency relation  $co$  and  $ES'$  an event structure with concurrency relation  $co'$ . Let  $\eta : ES \rightarrow ES'$  be a morphism of event structures. Then, for any events  $e_0, e_1$  of  $ES$ ,

$$e_0 \text{ } co \text{ } e_1 \text{ \& } \eta(e_0), \eta(e_1) \text{ both defined} \Rightarrow \eta(e_0) \text{ } co' \text{ } \eta(e_1).$$

Morphisms between event structures can be described more directly in terms of the causality and conflict relations of the event structure:

**Proposition 8.1.3.** A morphism of event structures from  $(E, \leq, \#)$  to  $(E', \leq', \#')$  is a partial function  $\eta : E \rightarrow_* E'$  such that

- (i)  $\eta(e)$  defined  $\Rightarrow [\eta(e)] \subseteq \eta[e]$  and
- (ii)  $\eta(e_0), \eta(e_1)$  both defined &  $\eta(e_0) \mathbb{W}' \eta(e_1) \Rightarrow e_0 \mathbb{W} e_1$ .

The category of event structures possesses products and coproducts useful in modelling parallel compositions and nondeterministic sums.

**Proposition 8.1.4.** *Let  $(E_0, \leq_0, \#_0)$  and  $(E_1, \leq_1, \#_1)$  be event structures. Their coproduct in the category **E** is the event structure  $(E_0 \uplus E_1, \leq, \#)$  where*

$$e \leq e' \Leftrightarrow (\exists e_0, e'_0. e_0 \leq_0 e'_0 \ \& \ j_0(e_0) = e \ \& \ j_0(e'_0) = e') \text{ or } \\ (\exists e_1, e'_1. e_1 \leq_1 e'_1 \ \& \ j_1(e_1) = e \ \& \ j_1(e'_1) = e')$$

and

$$\# = \#_0 \cup \#_1 \cup (j_0 E_0) \times (j_1 E_1),$$

with injections  $j_0 : E_0 \rightarrow E_0 \uplus E_1, j_1 : E_1 \rightarrow E_0 \uplus E_1$  the injections of  $E_0$  and  $E_1$  into their disjoint union.

It is tricky to give a direct construction of product on event structures. However, a construction of the product of event structures will follow from the product of trace languages and the coreflection from event structures to trace languages (see corollary 8.3.18), and we postpone the construction till then.

## 8.2 Domains of configurations

Viewing computation states as such subsets, progress in a computation is measured by the occurrence of more events. Let  $x, y \in \mathcal{D}(E)$  for an event structure  $E$ . If  $x \subseteq y$  then  $x$  can be regarded as a subbehaviour of  $y$ . The relation of inclusion between configurations is an information order of the sort familiar from denotational semantics, but special in that more information corresponds to more events having occurred. It is easy to see that the order  $(\mathcal{D}(E), \subseteq)$  has least upper bounds, when they exist, given as unions, and that the order is a complete partial order with least element the empty set. The domains associated with event structures turn out to be familiar. (Proofs of the following characterisations can be found in [Winskel, 1988b].)

The simplest characterisation of the domains represented by prime event structures starts by observing that an event  $e$  in an event structure corresponds to the configuration  $[e]$ . Such elements are characterised as being *complete primes* and domains of configurations have the property that every element is the least upper bound of these special elements.

**Definition 8.2.1.** Let  $(D, \sqsubseteq)$  be a partial order with least upper bounds of subsets  $X$  written as  $\bigsqcup X$  when they exist.

Say  $D$  is *bounded complete* iff all subsets  $X \subseteq D$  which have an upper bound in  $D$  have a least upper bound  $\bigsqcup X$  in  $D$ .

Say  $D$  is *coherent* iff all subsets  $X \subseteq D$  which are pairwise bounded (i.e. such that all pairs of elements  $d_0, d_1 \in X$  have upper bounds in  $D$ ) have least upper bounds  $\sqcup X$  in  $D$ . (Note that coherence implies bounded completeness.)

A *complete prime* of  $D$  is an element  $p \in D$  such that

$$p \subseteq \sqcup X \Rightarrow \exists x \in X. p \subseteq x$$

for any set  $X$  for which  $\sqcup X$  exists.

$D$  is *prime algebraic* iff

$$x = \sqcup \{p \subseteq x \mid p \text{ is a complete prime}\},$$

for all  $x \in L$ . If furthermore the sets

$$\{p \subseteq q \mid p \text{ is a complete prime}\}$$

are always finite when  $q$  is a complete prime, then  $D$  is said to be *finitary*.

If  $D$  is bounded complete and prime algebraic it is a *prime algebraic domain*.

**Theorem 8.2.2.** *Let  $E$  be an event structure. The partial order  $(\mathcal{D}(E), \subseteq)$  is a coherent, finitary, prime algebraic domain; the complete primes are the set  $\{[e] \mid e \in E\}$ .*

**Proof.** See [Nielsen *et al.*, 1981]. ■

Conversely, any coherent, finitary, prime algebraic domain is associated with an event structure in which the events are its complete primes.

**Theorem 8.2.3.** *Let  $(D, \subseteq)$  be a coherent, finitary, prime algebraic domain. Define  $(P, \leq, \#)$  to consist of  $P$ , the complete primes of  $D$ , ordered by*

$$p \leq p' \Leftrightarrow p \subseteq p',$$

*and with relation*

$$p \# p' \Leftrightarrow p \not\subseteq p',$$

*for  $p, p' \in P$ . Then  $(P, \leq, \#)$  is an event structure, with  $\phi : (D, \subseteq) \cong (\mathcal{D}(P, \leq, \#), \subseteq)$  giving an isomorphism of partial orders where  $\phi(d) = \{p \subseteq d \mid p \text{ is a complete prime}\}$  with inverse  $\theta : \mathcal{D}(P, \leq, \#) \rightarrow (D, \subseteq)$  given by  $\theta(x) = \sqcup x$ .*

**Proof.** See [Nielsen *et al.*, 1981]. ■

Event structures and coherent, finitary, prime algebraic domains are equivalent; one can be used to represent the other. Such domains are familiar in another guise. (Recall that the dI-domains of Berry are distributive



algebraic cpos in which every finite element only dominates finitely many elements [Berry, 1979].)

**Theorem 8.2.4.** *The finitary, prime algebraic domains are precisely the dI-domains of Berry.*

**Proof.** See [Winskel, 1988b] or [Winskel, 1983]. ■

Following Girard, call a function *linear* iff it is *stable* in the sense of Berry (i.e. preserves bounded meets) and preserves joins when they exist. The *covering* relation between configurations of an event structure is given by

$$x \rightarrow x' \Leftrightarrow_{def} \exists e. x \xrightarrow{e} x',$$

for configurations  $x, x'$ .

**Theorem 8.2.5.** *The category of event structures  $\mathbf{E}$  is equivalent to the subcategory of coherent dI-domains with morphisms those linear functions  $f : D \rightarrow D'$  which further satisfy*

$$x \rightarrow x' \Rightarrow f(x) = f(x') \text{ or } f(x) \rightarrow f(x').$$

**Proof.** A proof can be found in the report [Winskel, 1982]. ■

## 8.3 Event structures and trace languages

### 8.3.1 A representation theorem

Throughout this section assume  $(M, L, I)$  is a trace language. In this section we study the preorder

$$s \lesssim t \Leftrightarrow \exists u. su \approx t$$

of a trace language and show that its quotient  $\lesssim / \approx$  can be represented by the finite configurations of an event structure.

We use  $a, b, c, \dots$  for symbols in  $L$  and  $s, t, u, \dots$  for strings in  $L^*$ . Write  $N(b, s)$  for the number of occurrences of  $b$  in the string  $s$ . We write  $a \in s$  to mean  $a$  occurs in  $s$ , i.e.  $N(a, s) > 0$ . As an abbreviation, we write  $sIt$  if  $aIb$  for every symbol  $a$  in  $s$  and  $b$  in  $t$ .

Events of  $(M, L, I)$ , to be thought of as event occurrences, are taken to be equivalence classes of nonempty strings with respect to the equivalence relation  $\sim$  now defined.

**Definition 8.3.1.** The relation  $\sim$  is the smallest equivalence relation on nonempty strings such that

$$\begin{array}{lll} sa & \sim & sba \text{ if } bIa, \text{ and} \\ sa & \sim & ta \text{ if } s \approx t \end{array}$$

for  $sa, sba, ta \in M$ .

The next lemma yields an important technique for reasoning about trace languages.

**Lemma 8.3.2.** *Suppose  $sa, ta \in M$ .*

$$\neg(aIb) \ \& \ sa \sim ta \Rightarrow N(b, s) = N(b, t).$$

**Proof.** Assume  $\neg(aIb)$ . It is sufficient to verify the lemma's claim in the case of  $sa \sim ta$  where either

- (i)  $(t = sc \text{ and } cIa)$  or
- (ii)  $s \approx t$ .

If (i) then  $b \neq c$  (because one is independent of  $a$  and one not) so  $N(b, t) = N(b, sc) = N(b, s)$ . If (ii) then  $N(b, s) = N(b, t)$  because the number of occurrences of a symbol is invariant under  $\approx$ . ■

As  $\neg(aIa)$  the lemma in particular yields

$$sa \sim ta \Rightarrow N(a, s) = N(a, t)$$

for  $sa, ta \in M$ . Thus different occurrences of the same symbol in a string of  $M$  are associated with different events:

**Proposition 8.3.3.** *Suppose  $s_0a, s_1a$  are prefixes of  $t \in M$  such that  $s_0a \sim s_1a$ . Then  $s_0a = s_1a$ .*

We can now show how the preorder of trace languages coincides with the order of inclusion on the associated sets of events:

**Definition 8.3.4.** Let  $s \in M$ . Define the events of  $s$  to be

$$ev(s) = \{\{u\}_\sim \mid u \text{ is a nonempty prefix of } s\}.$$

**Lemma 8.3.5.** *Let  $s, t \in M$ . Then*

$$s \lesssim t \Leftrightarrow ev(s) \subseteq ev(t).$$

**Proof.** “ $\Rightarrow$ ”: We show the claim that, letting  $s, t \in M$ ,

$$s \approx t \Rightarrow ev(s) = ev(t),$$

from which “ $\Rightarrow$ ” follows. It is sufficient to establish this claim for the case where  $s = uabv$  and  $t = ubav$  with  $aIb$ . However, then

$$\begin{aligned} ev(s) &= ev(u) \cup \{\{ua\}_\sim, \{uab\}_\sim\} \cup \{\{uabv'\}_\sim \mid v' \text{ is a nonempty prefix of } v\} \\ &= ev(u) \cup \{\{uba\}_\sim, \{ub\}_\sim\} \cup \{\{ubav'\}_\sim \mid v' \text{ is a nonempty prefix of } v\} \\ &= ev(t). \end{aligned}$$

“ $\Leftarrow$ ”: This is proved by induction on  $s$ . The basis  $s = \varepsilon$  is obvious. Assume  $ev(s) \subseteq ev(t)$  where  $s = s'a$  and, inductively, that  $s' \lesssim t$ , i.e.  $s'u' \approx t$  for some  $u'$ . Because  $\{s'a\}_\sim \in ev(s)$  certainly  $\{s'a\}_\sim \in ev(t) = ev(s'u')$ . It follows that  $u' = u_0au_1$  for some  $u_0, u_1$  such that  $s'a \sim s'u_0a$ . (We cannot have  $\{s'a\}_\sim \in ev(s')$  by proposition 8.3.3 above.) We must have  $u_0Ia$  as otherwise there would be  $b \in u_0$  with  $\neg(bIa)$  and  $N(b, s') < N(b, s'u_0)$  contradicting lemma 8.3.2. Hence  $t \approx s'u_0au_1 \approx s'au_0u_1$  making  $s'a \lesssim t$ . This proves the induction step. ■

The next lemma shows that incompatibility between traces stems from a lack of independence between events.

**Lemma 8.3.6.** *Let  $s, t \in M$ .*

$$\exists u \in M. ev(u) = ev(s) \cup ev(t)$$

*iff*

$$\forall v \in M, a, b \in L. \{va\}_\sim \in ev(s) \ \& \ \{vb\}_\sim \in ev(t) \Rightarrow a(I \cup 1_L)b.$$

**Proof.** “if”: This implication is proved by induction on  $t$ . The basis case when  $t = \varepsilon$  is obvious. To prove the induction step assume  $t = t'a \in M$  and that

$$\{va\}_\sim \in ev(s) \ \& \ \{vb\}_\sim \in ev(t) \Rightarrow a = b \text{ or } aIb$$

for all  $v \in M$  and  $a, b \in L$ . Inductively we assume that there is  $u' \in M$  for which  $ev(u') = ev(s) \cup ev(t')$ . If  $\{t'a\}_\sim \in ev(s)$  then  $ev(u') = ev(s) \cup ev(t'a)$  as required. Assume otherwise that  $\{t'a\}_\sim \notin ev(s)$ . By lemma 8.3.5,  $t' \lesssim u'$  so  $t'w \approx u'$  for some string  $w$ . Assume  $w$  has the form  $b_1 \dots b_k$ . By lemma 8.3.5 and proposition 8.3.3, we necessarily have  $\{t'b_1 \dots b_i\}_\sim \in ev(s)$  for all  $i$  where  $0 < i \leq k$ . Let

$$u_i = t'b_1 \dots b_i a$$

for  $0 \leq i \leq k$ . We show by induction on  $i$  that  $u_i \in M$  and  $u_i \sim t'a$ . Certainly this holds for the basis when  $u_0 = t'a$ . To establish the induction step assume  $i > 0$  and, inductively, that  $u_{i-1} = t'b_1 \dots b_{i-1}a \in M$ . Because  $\{t'b_1 \dots b_{i-1}a\}_\sim \in ev(t)$  and  $\{t'b_1 \dots b_{i-1}b_i\}_\sim \in ev(s)$  by assumption we have  $a = b_i$  or  $aIb_i$ . However,  $a \neq b_i$  because otherwise  $u_i = t'b_1 \dots b_{i-1}b_i$  making  $\{t'a\}_\sim \in ev(s)$ , contrary to our assumption. Now that we know  $aIb_i$  the coherence axiom on trace languages ensures

$$u_i = t'b_1 \dots b_{i-1}b_i a \in M.$$

In addition

$$u_i = t'b_1 \dots b_{i-1}b_ia \sim t'b_1 \dots b_{i-1}a = u_{i-1} \sim t'a.$$

Thus by induction we have established that

$$u_i \in M \text{ and } u_i \sim t'a$$

for  $0 \leq i \leq k$ . In particular

$$u_k = t'b_1 \dots b_k a = u'a \in M \text{ and } u_k \sim t'a.$$

It follows that

$$\begin{aligned} ev(u_k) &= ev(u') \cup \{\{t'a\}_\sim\} \\ &= ev(s) \cup ev(t') \cup \{\{t'a\}_\sim\} \\ &= ev(s) \cup ev(t'a). \end{aligned}$$

We can thus maintain the induction hypothesis whether or not  $\{t'a\}_\sim \in ev(s)$ . This establishes the “if” direction of the lemma by induction.

“only if”: Assume that  $ev(u) = ev(s) \cup ev(t)$  for  $s, t, u \in M$  and that the “only if” direction fails to hold. That is, suppose  $\{va\}_\sim \in ev(s)$ ,  $\{vb\}_\sim \in ev(t)$   $a \neq b$ , and  $\neg(aIb)$  for  $v \in M$  and  $a, b \in L$ . Then either  $u = u_0au_1bu_2$  with  $va \sim u_0a$  and  $vb \sim u_0au_1b$ , or the symmetric case with  $a$  and  $b$  interchanged. In the former case we observe that

$$\begin{aligned} N(a, v) &= N(a, u_0) \quad \text{as } va \sim u_0a \text{ by lemma 8.3.2,} \\ &< N(a, u_0au_1). \end{aligned}$$

But  $N(a, v) = N(a, u_0au_1)$  as  $vb \sim u_0au_1b$  by lemma 8.3.2. This, and the similar contradictions obtained in the symmetric case, demonstrate the absurdity of our supposition, and thus the “only if” direction of the lemma. ■

**Remark 8.3.7.** The above lemma implies that the preorder  $\lesssim$  satisfies a finite form of coherence in the sense that any pairwise bounded finite subset has a least upper bound. The coherence axiom on trace languages was essential in proving the “if” direction of the equivalence. Without the coherence axiom, a finite form of bounded completeness can be demonstrated, *i.e.* a finite set with an upper bound has a least upper bound. More precisely it can be shown without use of the coherence axiom that

$$s \lesssim u \text{ \& } t \lesssim u \Rightarrow \exists v, w. u \approx vw \text{ \& } ev(v) = ev(s) \cup ev(t)$$

for all  $s, t, u \in M$ , from which the finite form of bounded completeness follows.

The following lemma says that each event has a  $\lesssim$ -minimum representative.

**Lemma 8.3.8.** *For all events  $e$  there is  $sa \in e$  such that*

$$\forall ta \in e. sa \lesssim ta.$$

**Proof.** We use a characterisation of  $\sim$  in the proof. Define

$$sa \prec_1 ta \text{ iff } (t = sb \ \& \ bIa) \text{ or } s \approx t$$

for  $sa, ta$  in  $M$ . Take  $\sim_1 =_{def} \prec_1 \cup \prec_1^{-1}$ . Then it is easily seen that  $\sim = (\sim_1)^*$ .

Let  $e$  be an event. Choose  $sa$  a  $\lesssim$ -minimal element of  $e$ . We show by induction on  $k$  that

$$sa(\sim_1)^k ta \Rightarrow sa \lesssim ta \quad (1)$$

for  $ta \in e$ . As  $\sim = (\sim_1)^*$  the lemma follows.

The basic case, where  $k = 0$ , holds trivially. Assume inductively that (1) holds for  $k$ . If  $sa(\sim_1)^{k+1}ta$  then  $sa(\sim_1)^k ua \sim_1 ta$  for some  $ua \in M$ . From the induction hypothesis we obtain

$$sa \lesssim ua.$$

If  $ua \prec_1 ta$  then  $(t = ub \ \& \ bIa)$  for some  $b$  or  $u \approx t$ , and in either case  $ua \lesssim ta$  giving  $sa \lesssim ta$ , as required to maintain the induction hypothesis. The rub comes if  $ua(\sim_1)^{-1}ta$  and this relation holds through  $u = tb$  and  $bIa$  for some  $b$ . Gathering facts, we see that

$$sa \sim tba \text{ and } sa \lesssim tba \text{ with } bIa$$

and that we require  $sa \lesssim ta$ .

By lemma 8.3.5 we get

$$ev(sa) \subseteq ev(tba) = ev(tab) = ev(ta) \cup \{\{tab\}_\sim\}.$$

Thus if  $\{tab\}_\sim \notin ev(s)$  we obtain  $ev(sa) \subseteq ev(ta)$  and hence the required  $sa \lesssim ta$ , by lemma 8.3.5.

We will show by contradiction that  $\{tab\}_\sim \notin ev(s)$ . Suppose otherwise, that  $\{tab\}_\sim \in ev(s)$ . Then

$$s = s_0bs_1 \text{ where } s_0b \sim tab.$$

Suppose  $c \in s_1$  and  $\neg(cIb)$ . Then

$$N(c, t) > N(c, s_0)$$



which is impossible. Consequently  $s_1Ib$ . Thus  $sa = s_0bs_1a \sim s_0s_1ba \sim s_0s_1a$ . The fact that  $s_0s_1ab \approx sa$  contradicts the  $\lesssim$ -minimality of  $sa$ . From this contradiction we deduce  $\{tab\}_\sim \notin ev(s)$  from which, as remarked, the required  $sa \lesssim ta$  follows. ■

The minimum representatives are used in defining the event structure associated with a trace language.

**Definition 8.3.9.** Let  $T = (M, L, I)$  be a trace language.

Define

$$tle(M, L, I) = (E, \leq, \#)$$

where

- $E$  is the set of events of  $(M, L, I)$ ,
- $\leq$  is a relation between events  $e, e'$  given by  $e \leq e'$  iff  $e \in ev(s)$  where  $sa$  is a minimum representative of  $e'$ , and
- the relation  $e \# e'$  holds between events iff

$$\exists e_0, e'_0. e_0 \#_0 e'_0 \ \& \ e_0 \leq e \ \& \ e'_0 \leq e'$$

where, by definition,

$$e_0 \#_0 e'_0 \text{ iff } \exists v, a, b. va \in e_0 \ \& \ vb \in e'_0 \ \& \ \neg(a(I \cup 1_L)b).$$

Furthermore, define  $\lambda_T : E \rightarrow L$  by taking  $\lambda_T(\{sa\}_\sim) = a$ . (From the definition of  $\sim$ , it follows that  $\lambda_T$  is well-defined as a function.)

**Proposition 8.3.10.** Let  $T = (M, L, I)$  be a trace language. Then the structure  $tle(T) = (E, \leq, \#)$  given by definition 8.3.9 is an event structure for which

$$\begin{aligned} e \leq e' & \quad \text{iff} \quad \forall s \in M. e' \in ev(s) \Rightarrow e \in ev(s) \\ e \# e' & \quad \text{iff} \quad \forall s \in M. e \in ev(s) \Rightarrow e' \notin ev(s). \end{aligned}$$

**Proof.** The required facts follow by considering minimum representatives of events. ■

We now present the representation theorem for trace languages. We write  $(M/\approx, \lesssim/\approx)$  for the partial order obtained by quotienting the pre-order  $\lesssim$  by its equivalence  $\approx$ .

**Theorem 8.3.11.** Let  $T = (M, L, I)$  be a trace language. Let  $tle(T) = (E, \leq, \#)$ . There is an order isomorphism

$$Ev : (M/\approx, \lesssim/\approx) \rightarrow \mathcal{D}^0(E, \leq, \#)$$

where  $Ev(\{s\}_\approx) = ev(s)$ .

Moreover, for  $s \in M$ ,  $x \in \mathcal{D}^0(E, \leq, \#)$ , and  $a \in L$ ,

$$(\exists e. ev(s) \xrightarrow{e} x \text{ in } \mathcal{D}^0(E, \leq, \#) \ \& \ \lambda_T(e) = a) \Leftrightarrow (sa \in M \ \& \ ev(sa) = x). \quad (\dagger)$$

**Proof.** Let  $s \in M$ . By the “only if” direction of lemma 8.3.6 it follows that  $ev(s)$  is a conflict-free subset of events. By lemma 8.3.8,  $ev(s)$  is downwards closed with respect to  $\leq$ . The fact that  $Ev$  is well-defined, one-to-one, order preserving, and reflecting follows from lemma 8.3.5. To establish that  $Ev$  is an isomorphism it suffices to check  $Ev$  is onto. To this end we first prove  $(\dagger)$ .

The “ $\Leftarrow$ ” direction of the equivalence  $(\dagger)$  follows directly, as follows. Assume  $sa \in M$ , and  $ev(sa) = x$ . Then taking  $e = \{sa\}_\sim$  yields an event for which  $ev(s) \xrightarrow{e} x$  and  $\lambda_T(e) = a$ . To show “ $\Rightarrow$ ”, assume  $ev(s) \xrightarrow{e} x$  and  $\lambda_T(e) = a$ . Let  $ta$  be a minimum representative of the event  $e$ . As  $x$  is downwards closed

$$ev(t) \subseteq ev(s).$$

Because  $x$  is conflict-free we meet the conditions of lemma 8.3.6 (“if” direction, with  $s$  for  $s$  and  $ta$  for  $t$ ) and obtain the existence of  $u \in M$  such that

$$ev(u) = ev(s) \cup ev(ta) = x.$$

Hence  $s \lesssim u$ , i.e.  $sw \approx u$  for some string  $w$ . But  $ev(s) \xrightarrow{e} ev(sw)$ , so  $w$  must be  $a$  with  $sa \in e$ .

Now a simple induction on the size of  $x \in \mathcal{D}^0(E, \leq, \#)$  shows that there exists  $s \in M$  for which  $ev(s) = x$ . From this it follows that  $Ev$  is onto, and consequently that  $Ev$  is an order isomorphism.  $\blacksquare$

The representation theorem for trace languages establishes a connection between trace languages and the *pomset* languages of Pratt [Pratt, 1986]. Via the representation theorem, each trace of a trace language  $T = (M, L, I)$  corresponds to a labelled partial order of events (a *partially ordered multiset* or *pomset*)—the partial order on events in the trace is induced by that of the event structure and the labelling function is  $\lambda_T$ . The trace language itself then corresponds to a special kind of pomset language; it is special chiefly because the concurrency relations in the pomsets arise from a single independence relation on the alphabet of labels, so consequently pomsets of traces have no *autoconcurrency*—no two concurrent events have the same label. (See [Bloom and Kwiatkowska, 1992], [Rozoy and Thiagarajan, 1991] and [Grabowski, 1981] for more details.)

Via the representation theorem we can see how to read the concurrency relation of an event structure in trace-language terms:

**Proposition 8.3.12.** *For a trace language  $T = (M, L, I)$  the construction  $tle(M, L, I)$  is an event structure in which the concurrency relation satisfies*

$$e \text{ co } e' \text{ iff } \exists va, vb \in M. va \in e \ \& \ vb \in e' \ \& \ aIb. \quad (\dagger)$$

**Proof.** We show  $(\dagger)$ .

“if”: Assume  $va \in e, vb \in e'$  with  $aIb$ . Then certainly  $va, vb$  are compatible as traces making  $\neg e \# e'$ . Moreover,  $e, e' \notin ev(v)$  (by e.g. proposition 8.3.3) ensuring neither  $e \leq e'$  nor  $e' \leq e$ , whence  $e \text{ co } e'$ .

“only if”: Assuming  $e \text{ co } e'$  there are distinct configurations  $x = ([e] \setminus \{e\}) \cup ([e'] \setminus \{e'\})$  and  $x_1 = x \cup \{e\}, x_2 = x \cup \{e'\}, y = x \cup \{e, e'\}$ . From the representation theorem 8.3.11 there is  $v \in M$  such that  $ev(v) = x$ . Assume  $\lambda_T(e) = a$  and  $\lambda_T(e') = b$ . By  $(\dagger)$  of the representation theorem, as  $x \xrightarrow{e} x_1$  we obtain  $va \in M$  with  $ev(va) = x_1$ . It follows that  $va \in e$ . Again by  $(\dagger)$  of the representation theorem, this time because  $x_1 \xrightarrow{e'} y$  we obtain  $vab \in M$  with  $ev(vab) = y$ . It follows that  $vab \in e'$ . Similarly, it can be shown that  $vb \in M$  and  $vb \in e'$ . Because both  $vab$  and  $vb$  are representatives of the event  $e'$ , it follows directly that  $vb \sim vab$ . If  $\neg(aIb)$  then lemma 8.3.2 would imply  $N(a, v) = N(a, va)$ . But this is clearly absurd, yielding  $aIb$ . We have produced the  $va, vb \in M$  required. ■

### 8.3.2 A coreflection

The representation theorem extends to a coreflection between the categories of event structures and trace languages.

**Definition 8.3.13.** Let  $ES$  be an event structure with events  $E$ . Define  $etl(ES)$  to be  $(M, E, co)$ , where  $s = e_1 \dots e_n \in M$  iff there is a chain

$$\emptyset \xrightarrow{e_1} x_1 \xrightarrow{e_2} x_2 \dots \xrightarrow{e_n} x_n$$

of configurations of  $ES$ . Let  $\eta$  be a morphism of event structures  $\eta : ES \rightarrow ES'$ . Define  $etl(\eta) = \eta$ .

**Proposition 8.3.14.**  *$etl$  is a functor  $\mathbf{E} \rightarrow \mathbf{TL}$ .*

**Proof.** The only nontrivial part of the proof is that showing that  $\eta$  is a morphism from  $etl(E)$  to  $etl(E')$  provided  $\eta$  is a morphism  $ES \rightarrow ES'$ . However, this follows from the proposition 8.1.2 and the observation that if a sequence of events  $s$  is associated with a chain of configurations in  $ES$  then  $\hat{\eta}(s)$  is associated with a chain of configurations in  $ES'$ . ■

The function  $\lambda_T$ , for  $T$  a trace language, will be the counit of the adjunction.

**Proposition 8.3.15.** *Let  $T = (M, L, I)$  be a trace language. Then,*

$$\lambda_T : etl \circ tle(T) \rightarrow T$$

is a morphism of trace languages.

**Proof.** Let  $e_1 e_2 \cdots e_n$  be a string in the trace language  $etl \circ tle(T)$ . Then there is a chain of configurations of the event structure  $tle(T)$

$$\emptyset \xrightarrow{e_1} \{e_1\} \xrightarrow{e_2} \{e_1, e_2\} \cdots \xrightarrow{e_n} \{e_1, e_2, \dots, e_n\}.$$

By repeated use of  $(\dagger)$  in the representation theorem 8.3.11, we obtain that  $\widehat{\lambda}_T(e_1 e_2 \cdots e_n) \in M$ . If  $e \text{ co } e'$ , for events  $e, e'$ , then by proposition 8.3.12, it follows directly that  $\lambda_T(e) I \lambda_T(e')$ . Thus  $\lambda_T : etl \circ tle(T) \rightarrow T$  is a morphism of trace languages. ■

**Lemma 8.3.16.** *Let  $ES = (E, \leq, \#)$  be an event structure, such that  $etl(ES) = (M, E, \text{co})$ . Let  $\lambda : etl(ES) \rightarrow T'$  be a morphism in **TL**. If  $\lambda(e)$  is defined then for all  $se, s'e \in M$*

$$\hat{\lambda}(se) \sim \hat{\lambda}(s'e)$$

in  $T' = (M', L', I')$ .

**Proof.** It suffices to consider the following two cases.

The first case is where we assume  $s = ue_0 e_1 v$  and  $s' = ue_1 e_0 v$  where  $u, v \in E^*$ ,  $e_0, e_1 \in E$ ,  $e_0 \text{ co } e_1$  in  $ES$ : in this case  $e_0$  and  $e_1$  are independent in  $etl(ES)$ . But then  $\lambda(e_0) I' \lambda(e_1)$  in  $T'$  if both defined (from properties of morphisms in **TL**), and hence  $\hat{\lambda}(ue_0 e_1 v) \sim \hat{\lambda}(ue_1 e_0 v)$  in  $T'$ .

The second case arises when  $s = s'e'$  for some  $e' \in E$  such that  $e \text{ co } e'$  in  $ES$ : in this case  $e$  and  $e'$  are independent in  $etl(ES)$ . But then  $\lambda(e) I' \lambda(e')$  in  $T'$  if  $\lambda(e')$  is defined and hence  $\hat{\lambda}(se) \sim \hat{\lambda}(s'e)$ . ■

**Theorem 8.3.17.** *Let  $T' = (M', L', I')$  be a trace language. Then the pair  $etl \circ tle(T')$ ,  $\lambda_{T'}$ , is cofree over  $T'$  with respect to the functor  $etl$ . That is, for any event structure  $ES$  and morphism  $\lambda : etl(ES) \rightarrow T'$  there is a unique morphism  $\eta : ES \rightarrow tle(T')$  such that  $\lambda = \lambda_{T'} \circ etl(\eta)$ .*

**Proof.** Let  $ES = (E, \leq, \#)$ ,  $tle(T') = (E', \leq', \#')$ , and  $etl(ES) = (M, E, \text{co})$ . Define  $\eta : E \rightarrow^* E'$  by

$$\eta(e) = \begin{cases} * & \text{if } \lambda(e) = * \\ \{\hat{\lambda}(se)\}_{\sim}, & \text{where } se \in M, \text{ if } \lambda(e) \neq *. \end{cases}$$

It follows from lemma 8.3.16 that  $\eta$  is a well-defined partial function from  $E$  to  $E'$ . We need to prove that

- (a)  $\eta$  is a morphism  $ES \rightarrow tle(T')$
- (b)  $\lambda = \lambda_{T'} \circ \eta$
- (c)  $\eta$  is unique satisfying (a) and (b).

(a): To prove (a), that  $\eta$  is a morphism, it suffices by proposition 8.1.3 to prove (i) and (ii) below.

(i) For every  $e \in E$ , if  $\eta(e)$  is defined then  $[\eta(e)] \subseteq \eta([e])$

Choose  $se \in M$  such that the occurrences in  $s$  equal  $[e]$  (in  $ES$ ). Assume  $x'a' \in M'$  such that

$$\{x'a'\}_\sim \subseteq \{\hat{\lambda}(se)\}_\sim \text{ in } tle(T'). \quad (*)$$

We have to prove the existence of  $e_0 \in [e]$  in  $E$  such that  $\{x'a'\}_\sim = \eta(e_0)$ . But from  $(*)$  we may choose a minimal prefix  $s_0e_0$  of  $se$  such that  $x'a' \sim \hat{\lambda}(s_0e_0)$ , with  $e_0 \in [e]$  from which we conclude the desired property.

(ii) For all  $e, e' \in E$ .  $\eta(e)w'\eta(e') \Rightarrow ewe'$

Suppose  $\neg ewe'$  and  $\eta(e), \eta(e')$  are both defined. There are essentially two cases to consider, one where  $e \text{ co } e'$  and the other where  $e < e'$  (or symmetrically  $e' < e$ ). Firstly assume  $e \text{ co } e'$  in  $ES$ . Then

$$eIe' \text{ in } etl(ES) \ \& \ se \in M \ \& \ se' \in M,$$

for some  $s \in M$ . Applying the morphism  $\lambda$ , we obtain

$$\lambda(e)I'\lambda(e') \text{ in } T' \ \& \ \hat{\lambda}(se) \in M' \ \& \ \hat{\lambda}(se') \in M.$$

But now

$$\eta(e) \text{ co } \eta(e') \text{ in } tle(T')$$

from proposition 8.3.12.

Secondly, assuming  $e < e'$  in  $ES$ , there are  $s, s' \in E^*$  such that

$$ses'e' \in M.$$

Applying  $\lambda$ ,

$$\hat{\lambda}(ses'e') \in M'.$$

Thus

$$\eta(e) \in ev(\hat{\lambda}(ses'e')) \ \& \ \eta(e') \in ev(\hat{\lambda}(ses'e')),$$

from which it follows that  $\neg\eta(e) \# \eta(e')$  in  $tle(T')$ , by proposition 8.3.10. Furthermore, from  $ses'e' \in M$ , we get  $\eta(e) \neq \eta(e')$ ; the assumption  $\eta(e) = \eta(e')$  implies  $\lambda(e) = \lambda(e')$ , but  $\hat{\lambda}(se) \sim \hat{\lambda}(ses'e')$  contradicts lemma 8.3.2. This completes the proof of (a).

(b): If  $\lambda(e) = *$  then  $\eta(e) = *$ , so  $(\lambda_{T'} \circ \eta)(e) = *$ . If  $\lambda(e)$  is defined then  $\eta(e) = \{\hat{\lambda}(se)\}_\sim$  for some  $se \in M$ . This implies  $\lambda_{T'}(\eta(e)) = \lambda(e)$  by the definition of  $\lambda_{T'}$ . Hence  $\lambda = \lambda_{T'} \circ \eta$ .

(c): We now show the uniqueness of  $\eta$ . Assume  $\eta'$  is any morphism from  $E$  to  $tle(T)$ , such that  $\lambda_{T'} \circ \eta' = \lambda$ . We want to show  $\eta(e) = \eta'(e)$  for all



$e \in E$ . Let  $x \xrightarrow{e} x \cup \{e\}$  in  $ES$ , and assume inductively that  $\eta$  and  $\eta'$  agree on all elements of  $x$ . Firstly, from the assumption  $\lambda_{T'} \circ \eta' = \lambda$  we get  $\eta'(e)$  defined iff  $\lambda(e)$  defined (since  $\lambda_{T'}$  is total) and hence iff  $\eta(e)$  defined. So, assume  $\eta'(e)$  is defined and equal to  $e'$ . Then  $\eta'(x) \xrightarrow{e'} \eta'(x \cup \{e\})$  in  $\text{tle}(T')$  (since  $\eta'$  is a morphism) and  $\lambda_{T'}(e') = \lambda(e)$ . However, from the representation theorem for trace languages, it follows that there is exactly one event in  $\text{tle}(T')$  satisfying these requirements—the one picked by  $\eta$ , and hence  $\eta(e) = \eta'(e)$ . ■

**Corollary 8.3.18.** *The operation  $\text{tle}$  on trace languages extends to a functor, right adjoint to  $\text{etl}$ , forming a coreflection  $\mathbf{E} \hookrightarrow \mathbf{TL}$ ; the functor  $\text{tle}$  sends the morphism  $\lambda : T \rightarrow T'$  to  $\eta : \text{tle}(T) \rightarrow \text{tle}(T')$  acting on events  $\{sa\}_\sim$  of  $\text{tle}(T)$  so that*

$$\eta(\{sa\}_\sim) = \begin{cases} * & \text{if } \lambda(a) \text{ undefined,} \\ \{\widehat{\lambda}(sa)\}_\sim & \text{if } \lambda(a) \text{ defined.} \end{cases}$$

**Proof.** It follows from theorem 8.3.17 that  $\text{tle}$  extends to a functor, acting as described, so that the pair of functors form an adjunction. From the proof of theorem 8.3.17, the unit of this adjunction at  $ES$  is the morphism

$$\eta : ES \rightarrow \text{tle} \circ \text{etl}(ES)$$

given by  $\eta(e) = \{se\}_\sim$ , where  $se$  is a possible sequence of events of  $ES$ . It is easy to see that  $\eta$  is an isomorphism with inverse  $\eta^{-1} : \text{tle} \circ \text{etl}(ES) \rightarrow ES$  such that

$$\eta^{-1}(\{se\}_\sim) = e. \quad \blacksquare$$

The coreflection expresses the sense in which the model of event structures “embeds” in the model of trace languages. Because of the coreflection we can restrict trace languages to those which are isomorphic to images of event structures under  $\text{etl}$  and obtain a full subcategory of trace languages equivalent to that of event structures.

The existence of a coreflection from event structures to trace languages has the important consequence of yielding an explicit product construction on event structures, which is not so easy to define directly. The product of event structures  $E_0$  and  $E_1$  can be obtained as

$$\text{tle}(\text{etl}(E_0) \times \text{etl}(E_1)),$$

that is by first regarding the event structures as trace languages, forming their product as trace languages, and then finally regarding the result as an event structure again. That this result is indeed a product of  $E_0$  and

$E_1$  follows because the right adjoint  $tle$  preserves limits and the unit of the adjunction is a natural isomorphism (*i.e.* from the coreflection). In a similar way we will be able to obtain the product of event structures from that of nets, asynchronous transition systems, or indeed more general event structures, from the coreflections between categories of event structures and these models.

### 8.3.3 A reflection

The existence of a *coreflection* from the category of event structures to the category of trace languages might seem surprising, at least when seen alongside the analogous interleaving models, where we might think of trace languages as the analogue of languages. There is a *reflection* from the category of languages to the category of synchronisation trees.

This mismatch can be reconciled by recalling the two ways of regarding trace languages (*cf.* the discussion of section 7.2). One is that where the alphabet of a trace language is thought of as consisting of events; this view is adopted in establishing the coreflection. Alternatively, the alphabet can be thought of as a set of labels. With the latter view a more correct analogy is:

- Labelled event structures generalise synchronisation trees.
- Trace languages generalise languages.

As we will see shortly, this analogy can be formalised in a diagram of adjunctions.

In order to define the appropriate category of labelled event structures we first define the category  $\mathbf{Set}_I$  of sets with independence. It consists of objects  $(L, I)$  where  $L$  is a set and  $I$ , the independence relation, is a binary, irreflexive relation on  $L$ , and morphisms  $(L, I) \rightarrow (L', I')$  to be partial functions  $\lambda : L \rightarrow_* L'$  which preserve independence in the sense that

$$aIb \ \& \ \lambda(a) \text{ defined} \ \& \ \lambda(b) \text{ defined} \ \Rightarrow \ \lambda(a)I'\lambda(b);$$

composition is that of partial functions.

The category of labelled event structures  $\mathcal{L}_I(\mathbf{E})$  has objects  $(ES, l : (E, co) \rightarrow (L, I))$  where  $ES$  is an event structure with events  $E$ , and concurrency relation  $co$ , and the labelling function  $l : (E, co) \rightarrow (L, I)$  is a total function in  $\mathbf{Set}_I$  (which therefore sends concurrent events to independent labels). We remark that one way an ordinary set  $L$  can be regarded as a set with independence is as  $(L, L \times L \setminus 1_L)$ . The restriction on labelling functions to such sets with independence amounts to the commonly used restriction of banning *autoconcurrency* [van Glabbeek, 1990a]. A morphism in  $\mathcal{L}_I(\mathbf{E})$  has the form

$$(\eta, \lambda) : (ES, l : (E, co) \rightarrow (L, I)) \rightarrow (ES', l' : (E', co') \rightarrow (L', I'))$$

and consists of a morphism of event structures

$$\eta : ES \rightarrow ES'$$

and a morphism

$$\lambda : (L, I) \rightarrow (L', I')$$

in  $\mathbf{Set}_I$  such that

$$l' \circ \eta = \lambda \circ l.$$

The right adjoint of a reflection is  $\mathcal{E} : \mathbf{TL} \rightarrow \mathcal{L}_I(\mathbf{E})$  defined as follows:

**Definition 8.3.19.** Let  $T = (M, L, I)$  be a trace language. Define  $\mathcal{E}(M, L, I)$  to be  $(E, \leq, \#, \lambda_T)$  where  $(E, \leq, \#)$  is the event structure  $\text{tle}(T)$  and  $\lambda_T : (E, co) \rightarrow (L, I)$  in  $\mathbf{Set}_I$  is given by the counit at  $T$  of the coreflection between (unlabelled) event structures and trace languages.

Let  $\lambda : T \rightarrow T'$  be a morphism of trace languages. Define  $\mathcal{E}(\lambda)$  to be  $(\text{tle}(\lambda), \lambda)$ .

In constructing a left adjoint we make use of a relabelling operation on trace languages, where the relabelling function preserves independence. The operation is described in the following proposition.<sup>6</sup>

**Proposition 8.3.20.** Let  $\lambda : (L', I') \rightarrow (L, I)$  be a morphism in  $\mathbf{Set}_I$ . Let  $(M', L', I')$  be a trace language. Define  $\lambda_!(M', L', I')$  to be  $(M, L, I)$  where  $M$  is the smallest prefix-closed,  $I'$ -closed, and coherent subset (as in the definition of trace languages) containing  $\hat{\lambda}M'$ . Then  $\lambda : (M', L', I') \rightarrow \lambda_!(M', L', I')$  is a morphism of trace languages.

Let  $(ES, l)$  be a labelled event structure in  $\mathcal{L}_I(\mathbf{E})$ . Under the coreflection the event structure  $ES$  can be regarded as a trace language  $\text{etl}(ES)$  over an alphabet consisting of its events. Proposition 8.3.20 provides a morphism

$$\text{etl}(ES) \rightarrow l_! \circ \text{etl}(ES),$$

used in defining the left adjoint of the reflection  $\mathcal{T} : \mathcal{L}_I(\mathbf{E}) \rightarrow \mathbf{TL}$ , given by:

**Definition 8.3.21.** Let  $(ES, l) \in \mathcal{L}_I(\mathbf{E})$ , where  $ES = (E, \leq, \#)$  and  $l : (E, co) \rightarrow (L, I)$  in  $\mathbf{Set}_I$ . Define

$$\mathcal{T}(ES, l) = l_! \circ \text{etl}(ES).$$

For  $(\eta, \lambda) : (ES, l) \rightarrow (ES', l')$ , a morphism of  $\mathcal{L}_I(\mathbf{E})$ , define  $\mathcal{T}(\eta, \lambda) = \lambda$ .

---

<sup>6</sup>In fact, proposition 8.3.20 shows how to construct cocartesian liftings of the functor projecting morphisms  $\lambda : (M', L', I') \rightarrow (M, L, I)$  in  $\mathbf{TL}$  to morphisms  $\lambda : (L', I') \rightarrow (L, I)$  in  $\mathbf{Set}_I$ .

The proof that  $\mathcal{E}$  and  $\mathcal{T}$  constitute a reflection uses the coreflection of section 8.3.2. It hinges on the pun in which a set of events is regarded simultaneously as a set of labels.

**Theorem 8.3.22.**  $\mathcal{E} : \mathbf{TL} \rightarrow \mathcal{L}_I(\mathbf{E})$  and  $\mathcal{T} : \mathcal{L}_I(\mathbf{E}) \rightarrow \mathbf{TL}$  are functions with  $\mathcal{T}$  left adjoint to  $\mathcal{E}$ . In fact, if  $T = (M, L, I)$  is a trace language then

$$1_L : \mathcal{T}\mathcal{E}(T) \rightarrow T$$

is the counit at  $T$  making the adjunction a reflection.

**Proof.** It is easy to check that  $\mathcal{E}, \mathcal{T}$  are functors. The fact that  $\mathcal{T}\mathcal{E}(T) = T$  follows from the representation theorem (theorem 8.3.11), and ensures that  $1_L : \mathcal{T}\mathcal{E}(T) \rightarrow T$  is a morphism in  $\mathbf{TL}$ , for any  $T \in \mathbf{TL}$  with labelling set  $L$ .

Let  $(ES, l) \in \mathcal{L}_I(\mathbf{E})$ , with  $l : (E, co) \rightarrow (L', I)$ , and  $T = (M, L, I) \in \mathbf{TL}$ . We show the cofreeness property, that for any  $\lambda : \mathcal{T}(ES, l) \rightarrow T$  there is a unique morphism  $(\eta, \lambda) : (ES, l) \rightarrow \mathcal{E}(T)$  such that

$$\begin{array}{ccc} T & \xleftarrow{1_L} & \mathcal{T}\mathcal{E}(T) \\ & \searrow \lambda & \uparrow \mathcal{T}(\eta, \lambda) = \lambda \\ & & \mathcal{T}(ES, l) \end{array}$$

commutes. This follows from a corresponding cofreeness property associated with the coreflection  $\mathbf{E} \hookrightarrow \mathbf{TL}$ , as we now show.

First note that the cocartesian morphism

$$l : etl(ES) \rightarrow l_! \circ etl(ES) = \mathcal{T}(ES)$$

composes with

$$\lambda : \mathcal{T}(ES) \rightarrow T$$

to yield a morphism

$$\lambda \circ l : etl(ES) \rightarrow T.$$

By definition,  $(\eta, \lambda) : (ES, l) \rightarrow \mathcal{E}(T) = (tle(T), \lambda_T)$  is a morphism in  $\mathcal{L}_I(\mathbf{E})$  iff  $\eta : ES \rightarrow tle(T)$  is a morphism in  $\mathbf{E}$  and  $\lambda_T \circ \eta = \lambda \circ l$ . This is equivalent to

$\eta : ES \rightarrow tle(T)$  is a morphism in  $\mathbf{E}$  such that the following diagram commutes:

$$\begin{array}{ccc} T & \xleftarrow{\lambda_T} & etl \circ tle(T) \\ & \searrow \lambda \circ l & \uparrow \eta = etl(\eta) \\ & & etl(ES) \end{array}$$

But the coreflection between  $\mathbf{E} \hookrightarrow \mathbf{TL}$  ensures the existence of a unique such  $\eta : ES \rightarrow tle(T)$ . ■

The reflection

$$\mathbf{L} \hookleftarrow \mathbf{S}$$

between the interleaving models of Hoare languages and synchronisation trees is paralleled by the reflection

$$\mathbf{TL} \hookleftarrow \mathcal{L}_I(\mathbf{E})$$

between the noninterleaving models of labelled event structures and Mazurkiewicz trace languages. The strings in the Hoare languages are generalised to Mazurkiewicz traces. We can view the relationship in another way which relates to Pratt's work. A Mazurkiewicz trace corresponds to a pomset which has no autoconcurrency (no two concurrent events share the same label). Mazurkiewicz trace languages correspond to particular kinds of pomset languages, ones which are, in particular, associated with a global independence relation between actions. (See [Bloom and Kwiatkowska, 1992] for a precise characterisation.)

In more detail we have these reflections and coreflections:



The vertical coreflections have not been explained. Their left adjoints identify a synchronisation tree with a labelled event structure (the events are arcs), and a language with a trace language. In both cases the independence relation is taken to be empty.

What model is to generalise both transition systems and labelled event structures? A suitable model would consist of labels attached to certain structures; a fitting structure should allow loops in the behaviour and have events on which it is possible to interpret a relation of independence. There are several candidates for the appropriate structures, and we turn now to consider one of the earliest used.

## 9 Petri nets

Petri nets are a well-known model of parallel computation. They come in several variants, and we choose one which fits well with the other models of computation we have described (a good all-round reference on Petri nets



is [Adv, 1987]). Roughly, a Petri net can be thought of as a transition system where, instead of a transition occurring from a single global state, an occurrence of an event is imagined to affect only the conditions in its neighbourhood. The independence of events becomes a derived notion; two events are independent if their neighbourhoods of conditions do not intersect. As the definition of a *Petri net* (or simply *net*) we take:

**Definition 9.0.1.** A *Petri net* consists of  $(B, M_0, E, pre, post)$  where

$B$  is a set of *conditions*, with *initial marking*  $M_0$  a nonempty subset of  $B$ ,

$E$  is a set of *events*, and

$pre : E \rightarrow \mathcal{P}ow(B)$  is the *precondition* map such that  $pre(e)$  is nonempty for all  $e \in E$ ,

$post : E \rightarrow \mathcal{P}ow(B)$  is the *postcondition* map such that  $post(e)$  is nonempty for all  $e \in E$ .

A Petri net comes with an initial marking consisting of a subset of conditions which are imagined to hold initially. Generally, a *marking*, a subset of conditions, formalizes a notion of global state by specifying those conditions which hold. Markings can change as events occur, precisely how being expressed by the transitions

$$M \xrightarrow{e} M'$$

events  $e$  determine between markings  $M, M'$ . In defining this notion it is convenient to extend events by an "idling event".

**Definition 9.0.2.** Let  $N = (B, M_0, E, pre, post)$  be a Petri net with events  $E$ .

Define  $E_* = E \cup \{*\}$ .

We extend the pre- and post-condition maps to  $*$  by taking

$$pre(*) = \emptyset, \quad post(*) = \emptyset.$$

**Notation 9.0.3.** Whenever it does not cause confusion we write  $\bullet e$  for the preconditions  $pre(e)$  and  $e^\bullet$  for the postconditions,  $post(e)$ , of  $e \in E_*$ . We write  $\bullet e^\bullet$  for  $\bullet e \cup e^\bullet$ .

**Definition 9.0.4.** Let  $N = (B, M_0, E, pre, post)$  be a net.

For  $M, M' \subseteq B$  and  $e \in E_*$ , define

$$M \xrightarrow{e} M' \text{ iff } \bullet e \subseteq M \ \& \ e^\bullet \subseteq M' \ \& \ M \setminus \bullet e = M' \setminus e^\bullet.$$

Say  $e_0, e_1 \in E_*$  are *independent* iff  $\bullet e_0 \cap e_1^\bullet = \emptyset$ .

A marking  $M$  of  $N$  is said to be *reachable* when there is a sequence of events, possibly empty,  $e_1, e_2, \dots, e_n$  such that

$$M_0 \xrightarrow{e_1} M_1 \xrightarrow{e_2} \dots \xrightarrow{e_n} M_n = M$$

in  $N$ .

There is an alternative characterisation of the transitions between markings induced by event occurrences:

**Proposition 9.0.5.** *Let  $N$  be a net with markings  $M, M'$  and event  $e$ . Then*

$$M \xrightarrow{e} M' \text{ iff } \begin{array}{l} (1) \quad \bullet e \subseteq M \text{ \& } e^\bullet \cap (M \setminus \bullet e) = \emptyset \text{ and} \\ (2) \quad M' = (M \setminus \bullet e) \cup e^\bullet. \end{array}$$

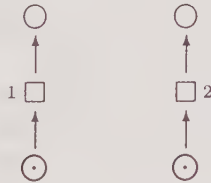
Property (1) expresses that the event  $e$  has *concession* at the marking  $M$ , while property (2) shows that the marking resulting from the occurrence of an event at a marking is unique.

We illustrate by means of a few small examples how nets can be used to model nondeterminism and concurrency. We make use of the commonly accepted graphical notations for nets in which events are represented by squares, conditions by circles, and the pre- and post-condition maps by directed arcs.

The holding of a condition is represented by marking it by a “token”; the distribution of tokens changes as the “token game” expressed in section 9.0.4 takes place.

### Example 9.0.6.

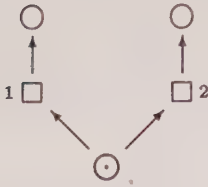
(1) Concurrency:



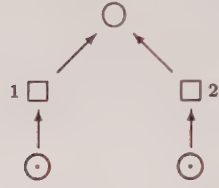
The events 1 and 2 can occur concurrently, in the sense that they both have concession and are independent in not having any pre- or post-conditions in common.

(2)

Forwards conflict:

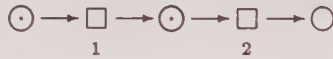


Backwards conflict:



Either one of events 1 and 2 can occur, but not both. This shows how nondeterminism can be represented in a net.

(3) Contact:



The event 2 has concession. The event 1 does not—its post-condition holds—and it can only occur after 2.

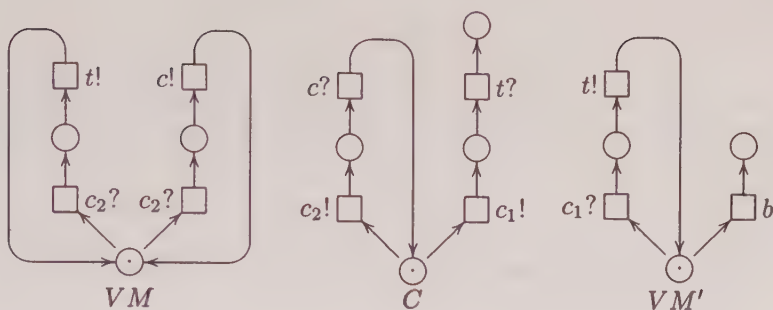
Example (3) above illustrates contact. In general, there is *contact* at a marking  $M$  when for some event  $e$

$${}^\bullet e \subseteq M \text{ \& } e^\bullet \cap (M \setminus {}^\bullet e) \neq \emptyset.$$

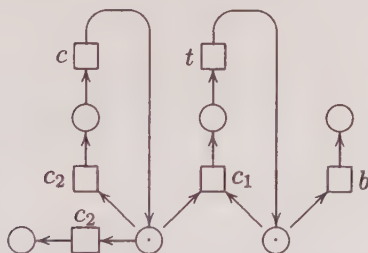
**Definition 9.0.7.** A net is said to be *safe* when contact never occurs at any reachable marking.

Many constructions on nets preserve safeness. As we shall see any net can be turned into a safe net with essentially the same behaviour—this follows from the coreflection between certain asynchronous transition systems and nets dealt with in section 10.2.2.

**Example 9.0.8.** We illustrate how one might model the customer-vending machine example of 3.3 by a net. We can represent its components by the following nets, in which the events are labelled:



Their composition as *SYS* can be represented by the following labelled net:



The fact that the *b*-event can occur concurrently (or in parallel with) either of the *c*<sub>2</sub>-events is reflected in the *b*-event and *c*<sub>2</sub>-event both having concession and being independent.

As this example makes clear, what's needed are operations on nets to build up this net description (or one with essentially the same behaviour). These will appear as universal constructions in the category of labelled nets.

## 9.1 A category of Petri nets

As morphisms on nets we take:<sup>7</sup>

**Definition 9.1.1.** Let  $N = (B, M_0, E, pre, post)$  and  $N' = (B', M'_0, E', pre', post')$  be nets. A *morphism*  $(\beta, \eta) : N \rightarrow N'$  consists of a relation  $\beta \subseteq B \times B'$ , such that  $\beta^{op}$  is a partial function  $B' \rightarrow B$ , and a partial function  $\eta : E \rightarrow E'$  such that

$$\begin{aligned} \beta M_0 &= M'_0, \\ \beta \bullet e &= \bullet \eta(e), \text{ and} \end{aligned}$$

<sup>7</sup>The morphisms on nets will preserve the transition-and-independence behaviour of nets while, as usual, respecting a choice of granularity fixed by the events. The rich structure of conditions on nets leaves room for variation, and another definition of morphism gives sensible results on the subclass of safe nets—see section 10.3.2.

$$\beta e^\bullet = \eta(e)^\bullet.$$

Thus morphisms on nets preserve initial markings and events when defined. A morphism  $(\beta, \eta) : N \rightarrow N'$  expresses how occurrences of events and conditions in  $N$  induce occurrences in  $N'$ . Morphisms on nets preserve behaviour:

**Proposition 9.1.2.** *Let  $N = (B, M_0, E, \text{pre}, \text{post})$ ,  $N' = (B', M'_0, E', \text{pre}', \text{post}')$  be nets. Suppose  $(\beta, \eta) : N \rightarrow N'$  is a morphism of net.*

- *If  $M \xrightarrow{e} M'$  in  $N$  then  $\beta M \xrightarrow{\eta(e)} \beta M'$  in  $N'$ .*
- *If  $\bullet e_1^\bullet \cap \bullet e_2^\bullet = \emptyset$  in  $N$  then  $\bullet \eta(e_1)^\bullet \cap \bullet \eta(e_2)^\bullet = \emptyset$  in  $N'$ .*

**Proof.** By definition,

$$\bullet \eta(e) = \beta^\bullet e \text{ and } \eta(e)^\bullet = \beta e^\bullet$$

for  $e$  an event of  $N$ . Observe too that because  $\beta^{op}$  is a partial function,  $\beta$  in addition preserves intersections and set differences. These observations mean that  $\beta M \xrightarrow{\eta(e)} \beta M'$  in  $N'$  follows from the assumption that  $M \xrightarrow{e} M'$  in  $N$ , and that independence is preserved. ■

**Proposition 9.1.3.** *Nets and their morphisms form a category in which the composition of two morphisms  $(\beta_0, \eta_0) : N_0 \rightarrow N_1$  and  $(\beta_1, \eta_1) : N_1 \rightarrow N_2$  is  $(\beta_1 \circ \beta_0, \eta_1 \circ \eta_0) : N_0 \rightarrow N_2$  (composition in the left component being that of relations and in the right that of partial functions).*

**Definition 9.1.4.** Let  $\mathbf{N}$  be the category of nets described above.

## 9.2 Constructions on nets

We examine some of the more important constructions in the category of nets. There are several constructions on nets which achieve the behaviour required of a nondeterministic sum of processes. We describe a coproduct in the category of nets.

**Definition 9.2.1.** Let  $N_0 = (B_0, M_0, E_0, \text{pre}_0, \text{post}_0)$  and  $N_1 = (B_1, M_1, E_1, \text{pre}_1, \text{post}_1)$  be nets. Define  $N_0 + N_1$  to be  $(B, M, E, \text{pre}, \text{post})$  where

$$B = M_0 \times M_1 \cup (B_0 \setminus M_0) \times_* (B_1 \setminus M_1),$$

which is associated with relations  $j_0 \subseteq B_0 \times B, j_1 \subseteq B_1 \times B$  given by

$$\begin{aligned} b_0 j_0 c &\Leftrightarrow \exists b_1 \in B_1 \cup \{*\}. c = (b_0, b_1) \\ b_1 j_1 c &\Leftrightarrow \exists b_0 \in B_0 \cup \{*\}. c = (b_0, b_1), \quad \text{and further} \\ M &= M_0 \times M_1, \\ E &= E_0 \uplus E_1, \text{ a disjoint union associated with injections} \end{aligned}$$



$in_0 : E_0 \rightarrow E_1, in_1 : E_1 \rightarrow E$ , and finally

$$pre(e) = j_0 \circ pre_0(e_0) \text{ and}$$

$$post(e) = j_0 \circ post_0(e_0) \text{ if } e = in_0(e_0), \text{ and}$$

$$pre(e) = j_1 \circ pre_1(e_1) \text{ and}$$

$$post(e) = j_1 \circ post_1(e_1) \text{ if } e = in_1(e_1).$$

The only peculiarity in this definition is the way in which the conditions are built. However, note that the relation  $\beta$  in any morphism

$$(\beta, \eta) : N \rightarrow N'$$

of nets  $N, N'$ , with conditions  $B, B'$  and initial markings  $M, M'$  respectively, corresponds to a pair of functions

$$\beta^{op} : (B' \setminus M') \rightarrow (B \setminus M) \text{ in } \mathbf{Set}_*,$$

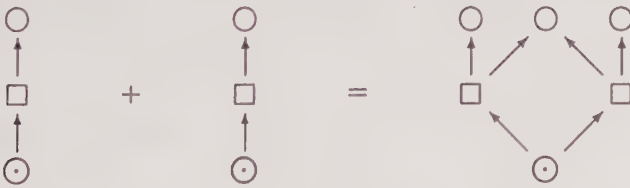
$$\beta^{op} : M' \rightarrow M \text{ in } \mathbf{Set}.$$

Thus it is to be expected that the conditions of a coproduct of nets correspond to products in  $\mathbf{Set}_* \times \mathbf{Set}$ . This remark handles the only obstacle in the proof of:

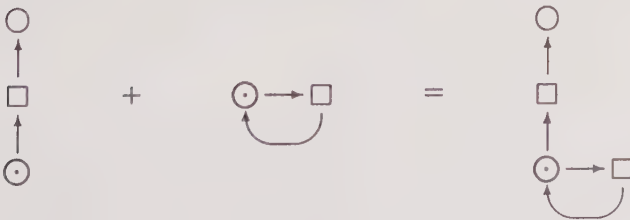
**Proposition 9.2.2.** *The construction  $N_0 + N_1$  above is a coproduct in the category of nets  $\mathbf{N}$  with injections  $(j_0, in_0) : N_0 \rightarrow N_0 + N_1, (j_1, in_1) : N_1 \rightarrow N_0 + N_1$ .*

**Example 9.2.3.**

(1)



(2)



In general the coproduct of nets can behave strangely, and allow a mix of behaviours from the two component nets. However, in the case where the component nets are safe, as they are in the example above, their coproduct is safe too and has a behaviour which can be described in terms of that of the components using the injection morphisms.

**Lemma 9.2.4.** *Suppose  $N_0, N_1$  are safe nets with initial markings  $M_0, M_1$  respectively. Then their coproduct  $N_0 + N_1$  is safe. Moreover:*

- (1a) *Two events  $in_0(e_0), in_0(e'_0)$  are independent in  $N_0 + N_1$  iff events  $e_0, e'_0$  are independent in  $N_0$ . Similarly, two events  $in_1(e_1), in_1(e'_1)$  are independent in  $N_0 + N_1$  iff events  $e_1, e'_1$  are independent in  $N_1$ .*  
 (1b) *Two events  $in_0(e_0), in_1(e_1)$  are independent in  $N_0 + N_1$  iff*

$$\begin{aligned} & \bullet e_0 \bullet \subseteq M_0 \text{ in } N_0 \quad \& \quad \bullet e_1 \bullet \cap M_1 = \emptyset \text{ in } N_1, \text{ or} \\ & \bullet e_1 \bullet \subseteq M_1 \text{ in } N_1 \quad \& \quad \bullet e_0 \bullet \cap M_0 = \emptyset \text{ in } N_0. \end{aligned}$$

(2)  *$X$  is reachable &  $X \xrightarrow{e} X'$  in  $N_0 + N_1$  iff*

$$\exists e_0, \text{ reachable } X_0, X'_0.$$

$$e = in_0(e_0) \& X_0 \xrightarrow{e_0} X'_0 \text{ in } N_0 \& X = j_0 X_0 \& X' = j_0 X'_0, \text{ or}$$

$$\exists e_1, \text{ reachable } X_1, X'_1.$$

$$e = in_1(e_1) \& X_1 \xrightarrow{e_1} X'_1 \text{ in } N_1 \& X = j_1 X_1 \& X' = j_1 X'_1.$$

**Proof.** (1a) is obvious, and (1b) follows from the way the conditions of the coproduct are constructed. The “if” direction of (2) follows as the injections are morphisms. The “only if” direction follows by showing: if  $X_0$  is a reachable marking of  $N_0$  and  $j_0 X_0 \xrightarrow{e} X'$  in  $N_0 + N_1$  then either

- (a)  $e = in_1(e_1) \& X_0 = M_0 \& X' = j_1 X'_1 \& M_1 \xrightarrow{e_1} X'_1$ , for some event  $e_1$  and marking  $X'_1$  of  $N_1$ , or  
 (b)  $e = in_0(e_0) \& X' = j_0 X'_0 \& X_0 \xrightarrow{e_0} X'_0$  in  $N_0$ , for some event  $e_0$  and marking  $X'_0$  of  $N_0$ .

To show this, assume  $j_0 X_0 \xrightarrow{e} X'$  in  $N_0 + N_1$  where  $X_0$  is a reachable marking of  $N_0$ . Consider first the case where  $e = in_1(e_1)$ . Because  $in_1(e_1)$  has concession at  $j_0 X_0$

$$\bullet in_1(e_1) \subseteq j_0 X_0$$

from which we see that

$$\bullet e_1 \subseteq M_1$$

—otherwise  $in_1(e_1)$  would have a precondition of the form  $(\bullet, b_1)$  which cannot be in the image  $j_0 X_0$  of the marking  $X_0$  of  $N_0$ . Because, by assumption, we have some  $b_1 \in \bullet e_1$  we see that

$$M_0 \times \{b_1\} \subseteq {}^\bullet in_1(e_1).$$

Because we now have

$$M_0 \times \{b_1\} \subseteq j_0 X_0$$

it follows that  $M_0 \subseteq X_0$ . But  $N_0$  is safe and  $X_0$  is assumed reachable from  $M_0$ , so we must have  $M_0 = X_0$ —otherwise a repetition of the same “token game” which led from  $M_0$  to  $X_0$ , but this time starting from  $X_0$ , would lead to contact. Letting  $X'_1$  be the marking such that  $M_1 \xrightarrow{e_1} X'_1$  we calculate

$$\begin{aligned} X' &= (j_0 X_0 \setminus {}^\bullet in_1(e_1)) \cup in_1(e_1)^\bullet \\ &= (M_0 \times M_1 \setminus {}^\bullet in_1(e_1)) \cup in_1(e_1)^\bullet \\ &= (j_1 M_1 \setminus j_1 {}^\bullet e_1) \cup j_1 e_1^\bullet \\ &= j_1 ((M_1 \setminus {}^\bullet e_1) \cup e_1^\bullet) \\ &= j_1 X'_1. \end{aligned}$$

This establishes (a) in the case where  $e = in_1(e_1)$ . In the other case, where  $e = in_0(e_0)$ , a similar but easier argument establishes (b). An analogous result holds for  $N_1$  in place of  $N_0$ . The “only if” direction of (2) now follows.

Suppose  $N_0 + N_1$  were not safe. Then

$${}^\bullet e \subseteq X \text{ \& } e^\bullet \cap (X \setminus {}^\bullet e) \neq \emptyset$$

for some reachable marking  $X$  and event  $e$  of  $N_0 + N_1$ . Suppose  $e = in_0(e_0)$ . Then by the results above, without loss of generality, we can suppose that  $X = j_0 X_0$  for some reachable marking  $X_0$  of  $N_0$ . By the definition of the pre- and post-conditions of events of  $N_0 + N_1$  we then obtain

$${}^\bullet e_0 \subseteq X_0 \text{ \& } e_0^\bullet \cap (X_0 \setminus {}^\bullet e_0) \neq \emptyset,$$

contradicting the assumption that  $N_0$  is safe. ■

The product of nets and its behaviour are more straightforward, and as is to be expected correspond to a synchronisation operation on nets.

**Definition 9.2.5.** Let  $N_0 = (B_0, M_0, E_0, pre_0, post_0)$  and  $N_1 = (B_1, M_1, E_1, pre_1, post_1)$  be nets. Their *product*  $N_0 \times N_1 = (B, E, M, pre, post)$ ; it has events

$$E = E_0 \times_* E_1,$$

the product in  $\mathbf{Set}_*$  with projections  $\pi_0 : E \rightarrow_* E_0$  and  $\pi_1 : E \rightarrow_* E_1$ . Its conditions have the form  $B = B_0 \uplus B_1$  the disjoint union of  $B_0$  and  $B_1$ . Define  $\rho_0$  to be the opposite relation to the injection  $\rho_0^{op} : B_0 \rightarrow B$ . Define  $\rho_1$  similarly. Take  $M = \rho_0^{op} M_0 + \rho_1^{op} M_1$  as the initial marking

of the product. Define the pre- and post-conditions of an event  $e$  in the product in terms of its pre- and post-conditions in the components by

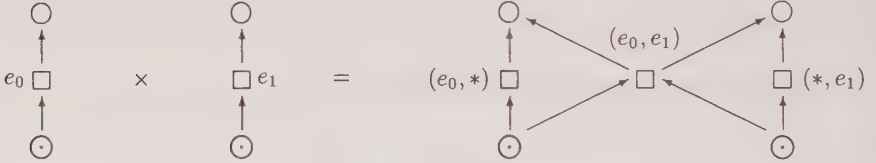
$$\begin{aligned} pre(e) &= \rho_0^{op}[pre_0(\pi_0(e))] + \rho_1^{op}[pre_1(\pi_1(e))] \\ post(e) &= \rho_0^{op}[post_0(\pi_0(e))] + \rho_1^{op}[post_1(\pi_1(e))]. \end{aligned}$$

**Proposition 9.2.6.** *The product  $N_0 \times N_1$ , with morphisms  $(\rho_0, \pi_0)$  and  $(\rho_1, \pi_1)$ , is a product in the category of Petri nets.*

**Proposition 9.2.7.** *The behaviour of a product of nets  $N_0 \times N_1$  is related to the behaviour of its components  $N_0$  and  $N_1$  by*

$$M \xrightarrow{e} M' \text{ in } N_0 \times N_1 \quad \text{iff} \\ (\rho_0 M \xrightarrow{\pi_0(e)} \rho_0 M' \text{ in } N_0 \ \& \ \rho_1 M \xrightarrow{\pi_1(e)} \rho_1 M' \text{ in } N_1).$$

**Example 9.2.8.** The product of two nets:



## 10 Asynchronous transition systems

Asynchronous transition systems deserve to be better known as a model of parallel computation. They were introduced independently by Bednarczyk in [Bednarczyk, 1988] and Shields in [Shields, 1985]. The idea on which they are based is simple enough: extend transition systems by, in addition, specifying which transitions are independent of which. More accurately, transitions are to be thought of as occurrences of events which bear a relation of independence. This interpretation is supported by axioms which essentially generalise those from Mazurkiewicz trace languages.

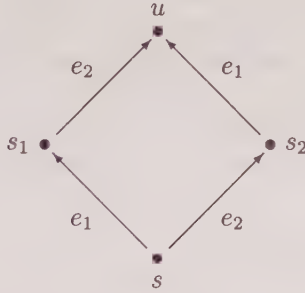
**Definition 10.0.1.** An *asynchronous transition system* consists of  $(S, i, E, I, Tran)$  where  $(S, i, E, Tran)$  is a transition system,  $I \subseteq E^2$ ; the *independence relation* is an irreflexive, symmetric relation on the set  $E$  of events such that

- (1)  $e \in E \Rightarrow \exists s, s' \in S. (s, e, s') \in Tran$
- (2)  $(s, e, s') \in Tran \ \& \ (s, e, s'') \in Tran \Rightarrow s' = s''$
- (3)  $e_1 I e_2 \ \& \ (s, e_1, s_1) \in Tran \ \& \ (s, e_2, s_2) \in Tran$

$$\Rightarrow \exists u. (s_1, e_2, u) \in \text{Tran} \ \& \ (s_2, e_1, u) \in \text{Tran}$$

$$(4) \quad e_1 I e_2 \ \& \ (s, e_1, s_1) \in \text{Tran} \ \& \ (s_1, e_2, u) \in \text{Tran} \\ \Rightarrow \exists s_2. (s, e_2, s_2) \in \text{Tran} \ \& \ (s_2, e_1, u) \in \text{Tran}.$$

Axiom (1) says that every event appears as a transition, and axiom (2) that the occurrence of an event at a state leads to a unique state. Axioms (3) and (4) express properties of independence: if two events can occur independently from a common state then they should be able to occur together and in so doing reach a common state (3); if two independent events can occur one immediately after the other then they should be able to occur with their order interchanged (4). Both situations lead to an “independence square” associated with the independence  $e_1 I e_2$ :



Axiom (3) corresponds to the coherence axiom on Mazurkiewicz trace languages, and, as there, a great deal of the theory can be developed without it.<sup>8</sup>

**Example 10.0.2.** We return to the example of 3.3. The transition system representing *SYS* in section 3.3 cannot be the underlying transition system of an asynchronous transition system with events identified with labels—there are, for example, distinct  $c_2$  transitions from the initial state. If, however, we distinguish between the two ways in which  $c_2$  can occur, and take the set of events to be

$$\{b, c, c_1, c'_2, c''_2, t\}$$

and explicitly assert the independencies

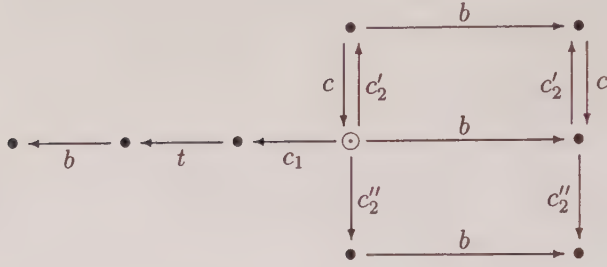
$$b I c'_2, \text{ and } b I c''_2,$$

we can obtain an asynchronous transition system with transitions

---

<sup>8</sup>Without axiom (3) asynchronous transition systems generate Mazurkiewicz trace languages, but without the coherence axiom, and unfold to more general event structures than those with a *binary* conflict.





The parallelism is caught by the independence squares (two upper and one lower).

Morphisms between asynchronous transition systems are morphisms between their underlying transition systems which preserve the additional relations of independence.

**Definition 10.0.3.** Let  $T = (S, i, E, I, Tran)$  and  $T' = (S', i', E', I', Tran')$  be asynchronous transition systems. A *morphism*  $T \rightarrow T'$  is a morphism of transition systems

$$(\sigma, \eta) : (S, i, E, Tran) \rightarrow (S', i', E', Tran')$$

such that

$$e_1 I e_2 \ \& \ \eta(e_1), \eta(e_2) \text{ both defined} \Rightarrow \eta(e_1) I' \eta(e_2).$$

Morphisms of asynchronous transition systems compose as morphisms between their underlying transition systems, and are readily seen to form a category.

**Definition 10.0.4.** Write **A** for the category of asynchronous transition systems.

The category **A** has categorical constructions which essentially generalise those of transition systems and Mazurkiewicz traces. Here are the product and coproduct constructions for the category **A**:

**Definition 10.0.5.** Assume asynchronous transition systems  $T_0 = (S_0, i_0, E_0, I_0, Tran_0)$  and  $T_1 = (S_1, i_1, E_1, I_1, Tran_1)$ . Their *product*  $T_0 \times T_1$  is  $(S, i, E, I, Tran)$  where  $(S, i, E, Tran)$  is the product of transition systems  $(S_0, i_0, E_0, Tran_0)$  and  $(S_1, i_1, E_1, Tran_1)$  with projections  $(\rho_0, \pi_0)$  and  $(\rho_1, \pi_1)$ , and the independence relation  $I$  is given by

$$\begin{aligned} a I b \Leftrightarrow & (\pi_0(a), \pi_0(b) \text{ defined} \Rightarrow \pi_0(a) I_0 \pi_0(b)) \ \& \\ & (\pi_1(a), \pi_1(b) \text{ defined} \Rightarrow \pi_1(a) I_1 \pi_1(b)). \end{aligned}$$

**Definition 10.0.6.** Assume asynchronous transition systems  $T_0 = (S_0, i_0, E_0, I_0, Tran_0)$  and  $T_1 = (S_1, i_1, E_1, I_1, Tran_1)$ . Their *coproduct*  $T_0 + T_1$  is  $(S, i, E, I, Tran)$  where  $(S, i, E, Tran)$  is the coproduct of transition systems  $(S_0, i_0, E_0, Tran_0)$  and  $(S_1, i_1, E_1, Tran_1)$  with injections  $(in_0, j_0)$  and  $(in_1, j_1)$ , and the independence relation  $I$  is given by

$$aIb \Leftrightarrow (\exists a_0, b_0. a = j_0(a_0) \ \& \ b = j_0(b_0) \ \& \ a_0 I_0 b_0) \text{ or } (\exists a_1, b_1. a = j_1(a_1) \ \& \ b = j_1(b_1) \ \& \ a_1 I_1 b_1).$$

## 10.1 Asynchronous transition systems and trace languages

That asynchronous transition systems generalise trace languages is backed up by a straightforward coreflection between categories of trace languages and asynchronous transition systems. To obtain the adjunction we need to restrict trace languages to those where every element of the alphabet occurs in some trace (this matches property (1) required by the definition of asynchronous transition systems).

**Definition 10.1.1.** Define  $\mathbf{TL}_0$  to be the full subcategory of trace languages  $(M, E, I)$  satisfying

$$\forall e \in E \exists s. se \in M.$$

A trace language forms an asynchronous transition system in which the states are traces.

**Definition 10.1.2.** Let  $(M, E, I)$  be a trace language in  $\mathbf{TL}_0$ , with trace equivalence  $\approx$ . Define  $tla(M, E, I) = (S, i, E, I, Tran)$  where

$$\begin{aligned} S &= M / \approx \text{ with } i = \{\epsilon\}_{\approx} \\ (t, e, t') \in Tran &\Leftrightarrow \exists s, se \in M. t = \{s\}_{\approx} \ \& \ t' = \{se\}_{\approx}. \end{aligned}$$

Let  $\eta : (M, E, I) \rightarrow (M', E', I')$  be a morphism of trace languages. Define  $tla(\eta) = (\sigma, \eta)$  where

$$\sigma(\{s\}_{\approx}) = \{\hat{\eta}(s)\}_{\approx}.$$

(Note this is well-defined because morphisms between trace languages respect  $\approx$ —this follows directly from proposition 7.1.6.)

**Proposition 10.1.3.** *The operation  $tla$  is a functor  $\mathbf{TL}_0 \rightarrow \mathbf{A}$ .*

An asynchronous transition system determines a trace language:

**Definition 10.1.4.** Let  $T = (S, i, E, I, Tran)$  be an asynchronous transition system. Define  $atl(T) = (Seq, E, I)$  where  $Seq$  consists of all strings of events, possibly empty,  $e_1 e_2 \dots e_n$  for which there are transitions

$$(i, e_1, s_1), (s_1, e_2, s_2), \dots, (s_{n-1}, e_n, s_n) \in \text{Tran}.$$

Let  $(\sigma, \eta) : T \rightarrow T'$  be a morphism of asynchronous transition systems. Define  $\text{atl}(\sigma, \eta) = \eta$ .

**Proposition 10.1.5.** *The operation  $\text{atl}$  is a functor  $\mathbf{A} \rightarrow \mathbf{TL}_0$ .*

In fact, the functors  $\text{tla}$ ,  $\text{atl}$  form a coreflection:

**Theorem 10.1.6.** *The functor  $\text{tla} : \mathbf{TL}_0 \rightarrow \mathbf{A}$  is left adjoint to  $\text{atl} : \mathbf{A} \rightarrow \mathbf{TL}_0$ .*

Let  $L = (M, E, I)$  be a trace language. Then  $\text{atl} \circ \text{tla} (M, E, I) = (M, E, I)$  and the unit of the adjunction at  $(M, E, I)$  is the identity  $1_E : (M, E, I) \rightarrow \text{atl} \circ \text{tla}(M, E, I)$ .

Let  $T$  be an asynchronous transition system, with events  $E$ . Then  $(\sigma, 1_E) : \text{tla} \circ \text{atl}(T) \rightarrow T$  is the counit of the adjunction at  $T$ , where  $\sigma(t)$ , for a trace  $t = \{e_1 e_2 \dots e_n\}_{\sim}$ , equals the unique state  $s$  for which  $i \xrightarrow{e_1 e_2 \dots e_n} s$ .

**Proof.** Let  $L = (M, E, I)$  be a trace language in  $\mathbf{TL}_0$  and  $T = (S, i, E', I', \text{Tran}')$  be an asynchronous system. Given a morphism of trace languages

$$\eta : L \rightarrow \text{atl}(T)$$

there is a unique morphism of asynchronous transition systems

$$(\sigma, \eta) : \text{tla}(L) \rightarrow T$$

—the function  $\sigma$  must act so  $\sigma(t)$ , on a trace  $t = \{e_1 e_2 \dots e_n\}_{\sim}$ , equals the unique state  $s_n$  for which there are transitions, possibly idle,

$$(i, \eta(e_1), s_1), (s_1, \eta(e_2), s_2), \dots, (s_{n-1}, \eta(e_n), s_n)$$

in  $T$ . That this is well-defined follows from  $T$  satisfying axiom (4) in the definition of asynchronous transition systems. The stated coreflection, and the form of the counit, follow. ■

The coreflection does not extend to an adjunction from  $\mathbf{TL}$  to  $\mathbf{A} - \mathbf{TL}_0$  is a reflective and not a coreflective subcategory of  $\mathbf{TL}$ .

We note that a coreflection between event structures and asynchronous transition systems follows by composing the coreflections between event structures and trace languages and that between trace languages and asynchronous transition systems. It is easy to see that the coreflection  $\mathbf{E} \hookrightarrow \mathbf{TL}$  restricts to a coreflection to  $\mathbf{E} \hookrightarrow \mathbf{TL}_0$ . The left adjoint of the resulting coreflection is the composition

$$\mathbf{E} \xrightarrow{\text{etl}} \mathbf{TL}_0 \xrightarrow{\text{tla}} \mathbf{A}.$$

A left adjoint of the coreflection can, however, be constructed more directly. The composition  $tla \circ etl$  is naturally isomorphic to the functor yielding an asynchronous transition system directly out of the configurations of the event structure, as is described in the next proposition.

**Proposition 10.1.7.** *For  $ES = (E, \leq, \#)$ , an event structure, define*

$$\Gamma(ES) = (\mathcal{D}^0(ES), \emptyset, E, co, Tran)$$

*where the transitions between configurations,  $Tran$ , consist of  $(x, e, x')$  where  $e \notin x$  &  $x' = x \cup \{e\}$ . For  $\eta : ES \rightarrow ES'$ , a morphism of event structures, define  $\Gamma(\eta) = (\sigma, \eta)$  where  $\sigma(x) = \eta x$ , for  $x$  a configuration of  $ES$ . This defines a functor  $\Gamma : \mathbf{E} \rightarrow \mathbf{A}$ . Moreover,  $\Gamma$  is naturally isomorphic to  $tla \circ etl$ .*

**Proof.** It is easy to check that  $\Gamma$  is a functor. The representation theorem 8.3.11, and its consequence, proposition 8.3.12, yield a morphism

$$(Ev^{-1}, \lambda_T) : \Gamma \circ tle(T) \rightarrow tla(T),$$

of asynchronous transition systems, which can be checked to be natural in  $T$ . Letting  $T$  be the trace language  $etl(ES)$ , of an event structure  $ES$ , we obtain a morphism

$$(Ev^{-1}, \lambda_{etl(ES)}) : \Gamma \circ tle \circ etl(ES) \rightarrow tla \circ etl(ES),$$

natural in  $ES$ . The coreflection 8.3.18 ensures that the counit at  $etl(ES)$

$$\lambda_{etl(ES)} : etl \circ tle \circ etl(ES) \rightarrow etl(ES)$$

is an isomorphism. This makes the function  $\lambda_{etl(ES)}$  a bijection, which together with the bijection  $Ev$  given by the representation theorem 8.3.11 ensures  $(Ev^{-1}, \lambda_{etl(ES)})$  is an isomorphism, necessarily natural in  $ES$ . It composes with the natural isomorphism  $\Gamma(\eta_{ES}) : \Gamma(ES) \rightarrow \Gamma \circ tle \circ etl(ES)$ , where  $\eta_{ES} : ES \rightarrow tle \circ etl(ES)$  is the unit of the coreflection at  $ES$ , to give the required natural isomorphism. ■

## 10.2 Asynchronous transition systems and nets

### 10.2.1 An adjunction

There is an adjunction between the categories  $\mathbf{A}$  and  $\mathbf{N}$ . First, note that there is an obvious functor from nets to asynchronous transition systems.

**Definition 10.2.1.** Let  $N = (B, M_0, E, \bullet, ( ), ( )^*)$  be a net. Define  $na(N) = (S, i, E, I, Tran)$  where

$S = \text{Pow}(B)$  with  $i = M_0$ ,

$e_1 I e_2 \Leftrightarrow \bullet e_1^\bullet \cap \bullet e_2^\bullet = \emptyset$ ,

$(M, e, M') \in \text{Tran} \Leftrightarrow M \xrightarrow{e} M'$  in  $N$ , for  $M, M' \in \text{Pow}(B)$ .

Let  $(\beta, \eta) : N \rightarrow N'$  be a morphism of nets. Define

$$na(\beta, \eta) = (\sigma, \eta)$$

where  $\sigma(M) = \beta M$ , for any  $M \in \text{Pow}(B)$ .

**Proposition 10.2.2.** *na is a functor  $\mathbf{N} \rightarrow \mathbf{A}$ .*

**Proof.** Letting  $N$  be a net, it is easily checked that  $na(N)$  is an asynchronous transition system: properties (1) and (2) of definition 10.0.1 are obvious while properties (3) and (4) follow directly from the interpretation of independence of events  $e_1, e_2$  as  $\bullet e_1^\bullet \cap \bullet e_2^\bullet = \emptyset$ . Letting  $(\beta, \eta) : N \rightarrow N'$  be a morphism of nets, proposition 9.1.2 yields that  $na(\beta, \eta)$  is a morphism  $na(N) \rightarrow na(N')$ . Clearly  $na$  preserves composition and identities. ■

As a preparation for the definition of a functor from asynchronous transition systems to nets we examine how a condition of a net  $N$  can be viewed as a subset of states and transitions of the asynchronous transition system  $na(N)$ . Intuitively the *extent*  $|b|$  of a condition  $b$  of a net is to consist of those markings and transitions at which  $b$  holds uninterruptedly. In fact, for simplicity, the extent  $|b|$  of a condition  $b$  is taken to be a subset of  $\text{Tran}_*$ , the transitions  $(M, e, M')$  and idle transitions  $(M, *, M)$  of  $na(N)$ ; the idle transitions  $(M, *, M)$  play the role of markings  $M$ .

**Definition 10.2.3.** Let  $b$  be a condition of a net  $N$ . Let  $\text{Tran}$  be the transition relation of  $na(N)$ . Define the *extent* of  $b$  to be

$$|b| = \{(M, e, M') \in \text{Tran}_* \mid b \in M \ \& \ b \in M' \ \& \ b \notin \bullet e^\bullet\}.$$

Not all subsets of transitions  $\text{Tran}_*$  of a net  $N$  are extents of conditions of  $N$ . For example, if  $(M, e, M') \notin |b|$  and  $(M', *, M') \in |b|$  for a transition  $M \xrightarrow{e} M'$  in  $N$  this means that the transition starts the holding of  $b$ . But then  $b \in e^\bullet$  so any other transition  $P \xrightarrow{e} P'$  must also start the holding of  $b$ . Of course, a condition cannot be started or ended by two independent events because, by definition, they can have no pre- or postcondition in common. These considerations motivate the following definition of the condition of a general asynchronous transition system.

**Definition 10.2.4.** Let  $T = (S, i, E, I, \text{Tran})$  be an asynchronous transition system. Its *conditions* are nonempty subsets  $b \subseteq \text{Tran}_*$  such that

$$(1) \ (s, e, s') \in b \Rightarrow (s, *, s) \in b \ \& \ (s', *, s') \in b$$



- (2) (i)  $(s, e, s') \in \bullet b \ \& \ (u, e, u') \in Tran \Rightarrow (u, e, u') \in \bullet b$   
(ii)  $(s, e, s') \in b^\bullet \ \& \ (u, e, u') \in Tran \Rightarrow (u, e, u') \in b^\bullet$

where for  $(s, e, s') \in Tran$  we define

$$\begin{aligned} (s, e, s') \in \bullet b &\Leftrightarrow (s, e, s') \notin b \ \& \ (s', *, s') \in b, \\ (s, e, s') \in b^\bullet &\Leftrightarrow (s, *, s) \in b \ \& \ (s, e, s') \notin b, \quad \text{and} \\ \bullet b^\bullet &= \bullet b \cup b^\bullet. \end{aligned}$$

- (3)  $(s, e_1, s') \in \bullet b^\bullet \ \& \ (u, e_2, u') \in \bullet b^\bullet \Rightarrow \neg e_1 I e_2.$

Let  $B$  be the set of conditions of  $T$ . For  $e \in E_*$ , define

$$\begin{aligned} e^\bullet &= \{b \in B \mid \exists s, s'. (s, e, s') \in \bullet b\}, \\ \bullet e &= \{b \in B \mid \exists s, s'. (s, e, s') \in b^\bullet\}, \text{ and} \\ \bullet e^\bullet &= \bullet e \cup e^\bullet. \end{aligned}$$

(Note that  $\bullet \bullet = \emptyset$ .)

Further, for  $s \in S$ , define  $M(s) = \{b \in B \mid (s, *, s) \in b\}$ .

As an exercise, we check that the extent of a condition of a net is indeed a condition of its asynchronous transition system.

**Lemma 10.2.5.** *Let  $N$  be a net with a condition  $b$ . Its extent  $|b|$  is a condition of  $na(N)$ . Moreover,*

- (I)  $(M, e, M') \in \bullet |b| \Leftrightarrow b \in e^\bullet$   
(II)  $(M, e, M') \in |b|^\bullet \Leftrightarrow b \in \bullet e$

whenever  $M \xrightarrow{e} M'$  in  $N$ .

**Proof.** We prove (I) (the proof of (II) is similar):

$$\begin{aligned} (M, e, M') \in \bullet |b| &\Leftrightarrow (M, e, M') \notin |b| \ \& \ (M', *, M') \in |b| \\ &\Leftrightarrow \neg(b \in M \ \& \ b \in M' \ \& \ b \notin \bullet e^\bullet) \ \& \ b \in M' \\ &\Leftrightarrow (b \notin M \ \& \ b \in M') \text{ or } (b \in \bullet e^\bullet \ \& \ b \in M') \\ &\Leftrightarrow b \in e^\bullet, \text{ as } M \xrightarrow{e} M'. \end{aligned}$$

Using (I) and (II), it is easy to check that  $|b|$  is a condition of  $na(N)$ . First we note that  $|b|$  is nonempty because it contains for instance  $(\{b\}, *, \{b\})$ .

We quickly run through the axioms required by definition 10.2.4:

- (1) If  $(M, e, M') \in |b|$  then  $b \in M$  and  $b \in M'$  whence  $(M, *, M), (M', *, M') \in |b|$ .  
(2) (i) If  $(M, e, M') \in \bullet |b|$  then  $b \in e^\bullet$ , by (I) “ $\Rightarrow$ ”. Hence, if  $P \xrightarrow{e} P'$  by (I) “ $\Leftarrow$ ” we obtain  $(P, e, P') \in \bullet |b|$ . The proof of (2)(ii) is similar.  
(3) (i) If  $(M, e_1, M'), (P, e_2, P') \in \bullet |b|$  then  $b \in e_1^\bullet$  and  $b \in e_2^\bullet$ , by (I) applied twice. Hence  $\neg e_1 I e_2$ .

**Definition 10.2.6.** Let  $(\sigma, \eta) : T \rightarrow T'$  be a morphism between asynchronous transition systems  $T = (S, i, E, I, \text{Tran})$  and  $T' = (S', i', E', I', \text{Tran}')$ . For  $b \subseteq \text{Tran}'_*$ , define

$$(\sigma, \eta)^{-1}b = \{(s, e, s') \in \text{Tran}_* \mid (\sigma(s), \eta(e), \sigma(s')) \in b\}.$$

**Lemma 10.2.7.** Let  $(\sigma, \eta) : T \rightarrow T'$  be a morphism between asynchronous transition systems. Let  $b$  be a condition of  $T'$ . Then  $(\sigma, \eta)^{-1}b$  is a condition of  $T$  provided it is nonempty. Furthermore,

$$(1) (\sigma, \eta)^{-1}b \in \bullet e \Leftrightarrow b \in \bullet \eta(e)$$

$$(2) (\sigma, \eta)^{-1}b \in e^\bullet \Leftrightarrow b \in \eta(e)^\bullet$$

for any event  $e$  of  $T$ .

**Proof.** We show (1), assuming  $b \subseteq \text{Tran}'_*$  and an event  $e$  of  $T$ . Observe that

$$\begin{aligned} (\sigma, \eta)^{-1}b \in \bullet e &\Leftrightarrow (s, e, s') \in (\sigma, \eta)^{-1}b^\bullet, \text{ for some states } s, s' \\ &\Leftrightarrow (s, *, s) \in (\sigma, \eta)^{-1}b \text{ \& } (s, e, s') \notin (\sigma, \eta)^{-1}b \\ &\Leftrightarrow (\sigma(s), *, \sigma(s)) \in b \text{ \& } (\sigma(s), \eta(e), \sigma(s')) \notin b \\ &\Leftrightarrow (\sigma(s), \eta(e), \sigma(s')) \in b^\bullet \\ &\Leftrightarrow b \in \bullet \eta(e). \end{aligned}$$

The proof of (2) is similar. That  $(\sigma, \eta)^{-1}b$  is a condition of  $T$ , if nonempty, follows straightforwardly from the assumption that  $b$  is a condition. ■

**Definition 10.2.8.** Let  $T = (S, i, E, I, \text{Tran})$  be an asynchronous transition system. Define  $\text{an}(T) = (B, M_0, E, \text{pre}, \text{post})$  by taking  $B$  to be the set of conditions of  $T$ ,  $M_0 = M(i)$ , with pre- and post-condition maps given by the corresponding operations in  $T$ , i.e.  $\text{pre}(e) = \bullet e$  and  $\text{post}(e) = e^\bullet$  in  $T$ . Let  $(\sigma, \eta) : T \rightarrow T'$  be a morphism of asynchronous transition systems. Define  $\text{an}(\sigma, \eta) = (\beta, \eta)$  where for conditions  $b$  of  $T$  and  $b'$  of  $T'$  we take

$$b\beta b' \text{ iff } b = (\sigma, \eta)^{-1}b'.$$

(Note that because of lemma 10.2.7,

$$b\beta b' \text{ iff } \emptyset \neq b = (\sigma, \eta)^{-1}b'$$

where we only assume  $b'$  is a condition of  $T'$ .)

The verification that  $\text{an}(T)$  is indeed a net involves demonstrating that every event has at least one pre- and post-condition. This follows from the following lemma which indicates how rich an asynchronous transition

system is in conditions (it says an arbitrary pairwise-dependent set of events can be made to be both the starting and ending events of a single condition):

**Lemma 10.2.9.** *Let  $T = (S, i, E, I, \text{Tran})$  be an asynchronous transition system. Suppose  $X$  is a nonempty subset of  $E$  such that*

$$e_1, e_2 \in X \Rightarrow \neg e_1 I e_2.$$

*Then, there is a condition  $b$  of  $T$  such that*

$$X = \{e \mid b \in e^\bullet\} \ \& \ X = \{e \mid b \in {}^\bullet e\}.$$

**Proof.** Define

$$b = \{(s, e, s') \in \text{Tran}_* \mid e \notin X\}.$$

It is simply checked that  $b$  is a condition with beginning and ending events  $X$ . ■

**Lemma 10.2.10.** *Let  $T = (S, i, E, I, \text{Tran})$  be an asynchronous transition system. Then  $\text{an}(T)$  is a net. Moreover,*

$$e_1 I e_2 \Leftrightarrow {}^\bullet e_1^\bullet \cap {}^\bullet e_2^\bullet = \emptyset,$$

and

$$(s, e, s') \in \text{Tran} \Rightarrow M(s) \xrightarrow{e} M(s') \text{ in } \text{an}(T).$$

**Proof.** For  $\text{an}(T)$  to be a net it is required that its initial marking, and pre- and post-conditions of events, be nonempty. However, taking  $b = \text{Tran}_*$  yields a condition in the initial marking, while for an event  $e$ , letting  $X$  be  $\{e\}$  in lemma 10.2.9 produces a pre- and post-condition of  $e$ .

If  $e_1 I e_2$  then axiom (3) on conditions (definition 10.2.4) ensures  ${}^\bullet e_1^\bullet \cap {}^\bullet e_2^\bullet = \emptyset$ . Conversely, by lemma 10.2.9, if  $\neg(e_1 I e_2)$  we can obtain a condition in  ${}^\bullet e_1^\bullet \cap {}^\bullet e_2^\bullet$ .

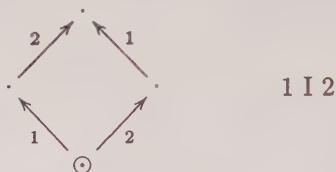
Suppose  $(s, e, s') \in \text{Tran}$ . Then, letting  $B$  be the set of conditions of  $T$ ,

$$\begin{aligned} {}^\bullet e &= \{b \in B \mid (s, *, s) \in b \ \& \ (s, e, s') \notin b\} \subseteq M(s), \\ e^\bullet &= \{b \in B \mid (s, e, s') \notin b \ \& \ (s', *, s') \in b\} \subseteq M(s'), \text{ and} \\ M(s) \setminus {}^\bullet e &= \{b \in B \mid (s, *, s) \in b\} \setminus \{b \in B \mid (s, *, s) \in b \ \& \ (s, e, s') \notin b\} \\ &= \{b \in B \mid (s, e, s') \in b\} \\ &= \{b \in B \mid (s', *, s') \in b\} \setminus \{b \in B \mid (s, e, s') \notin b \ \& \ (s', *, s') \in b\} \\ &= M(s') \setminus e^\bullet. \end{aligned}$$

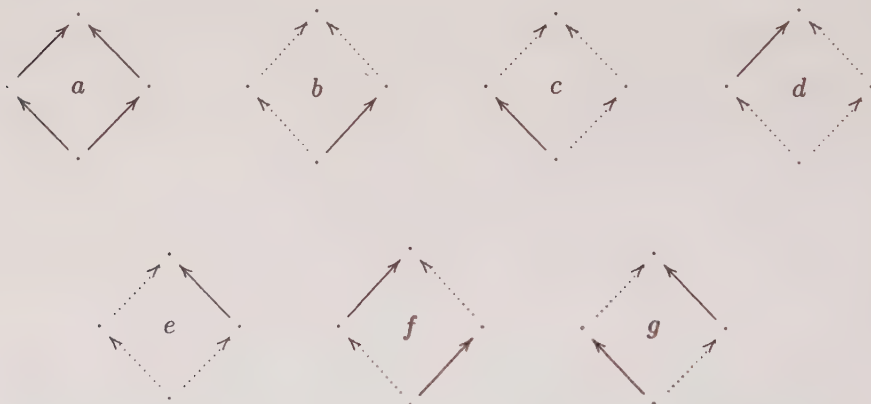
Thus  $M(s) \xrightarrow{e} M(s')$ . ■

We illustrate how a net is produced from an asynchronous transition system.

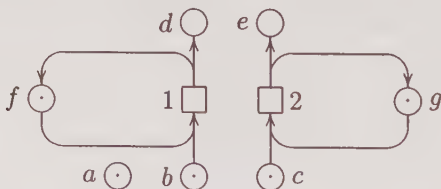
**Example 10.2.11.** Consider the following asynchronous transition system  $T$  with two independent events 1 and  $2\Lambda$ :



It has these conditions, where those transitions in the condition are represented by solid arrows:



Consequently the asynchronous transition system  $T$  yields this net  $an(T)$ :



**Lemma 10.2.12.**  $an$  is a functor  $\mathbf{A} \rightarrow \mathbf{N}$ .

**Proof.** The only difficulty comes in showing the well-definedness of  $an$  on morphisms. Let  $(\sigma, \eta) : T \rightarrow T'$  be a morphism of asynchronous transition systems  $T = (S, i, E, I, Tran)$ ,  $T' = (S', i', E', I', Tran')$ . We require that  $an(\sigma, \eta) =_{def} (\beta, \eta)$  is a morphism of nets  $an(T) \rightarrow an(T')$ . Let  $an(T) = (B, M_0, E, pre, post)$ ,  $an(T') = (B', M'_0, E', pre', post')$ . We see that  $\beta$  preserves initial markings by arguing:

$$\begin{aligned}
b' \in M'_0 &\Leftrightarrow (i', *, i') \in b' \\
&\Leftrightarrow (\sigma(i), *, \sigma(i)) \in b' \\
&\Leftrightarrow (i, *, i) \in (\sigma, \eta)^{-1}b' \\
&\Leftrightarrow \beta^{op}(b') \in M_0.
\end{aligned}$$

The fact that  $\beta e^\bullet = \bullet \eta(e)$  and  $\beta^\bullet e = \eta(e)^\bullet$  follows directly from (1) and (2) of lemma 10.2.7. ■

In fact,  $an$  is left adjoint to  $na$ . Before proving this we explore the unit and counit of the adjunction. The unit of the adjunction:

**Lemma 10.2.13.** *Let  $T$  be an asynchronous system. Defining  $\sigma_0(s) = M(s)$  for  $s$  a state of  $T$  and letting  $1_E$  be the identity on the events of  $T$ , we obtain a morphism of asynchronous transition systems*

$$(\sigma_0, 1_E) : T \rightarrow na \circ an(T).$$

**Proof.** That  $(\sigma_0, 1_E)$  is a morphism follows directly from lemma 10.2.10. ■

The counit:

**Lemma 10.2.14.** *Let  $N = (B, M_0, E, \bullet(\cdot), (\cdot)^\bullet)$  be a net. Let  $Tran$  be the transitions of  $na(N)$ . For  $b \in B$  and  $c \subseteq Tran_*$ , taking*

$$c\beta_0 b \Leftrightarrow_{def} c = |b|$$

*defines a relation between conditions of  $na(N)$  and  $B$ , such that*

$$(\beta_0, 1_E) : an \circ na(N) \rightarrow N$$

*is a morphism of nets.*

**Proof.** By lemma 10.2.5,  $|b|$  is a condition of  $na(N)$  if  $b$  is a condition of  $N$ . This ensures that  $\beta_0$  is a relation between the conditions of  $na(N)$  and  $B$ . We should check that  $(\beta_0, 1_E) : an \circ na(N) \rightarrow N$  is a morphism of nets. Let  $M'_0$  be the initial marking of  $an \circ na(N)$ . We see for any  $b \in B$  that

$$\begin{aligned}
\beta_0^{op}(b) \in M'_0 &\Leftrightarrow (M_0, *, M_0) \in \beta_0^{op}(b) \\
&\text{by the definition of } an \text{ and } na, \\
&\Leftrightarrow b \in M_0 \text{ by the definition of } \beta_0.
\end{aligned}$$

From the equivalence

$$\beta_0^{op}(b) \in M'_0 \Leftrightarrow b \in M_0$$



we deduce  $\beta_0 M_0 = M'_0$ , that  $\beta_0$  preserves initial markings. In addition  $\beta_0$  preserves pre- and post-conditions of events from (II), (I) of lemma 10.2.5. ■

Now we establish the adjunction between **A** and **N** in which  $an$  is left adjoint to  $na$ .

**Lemma 10.2.15.** *Let  $T = (S, i, E, I, Tran)$  be an asynchronous transition system and  $N = (B, M_0, E', pre, post)$  a net.*

*For a morphism of nets  $(\beta, \eta) : an(T) \rightarrow N$ , defining  $\sigma(s) = \beta \circ M(s)$ , for  $s \in S$ , yields a morphism of asynchronous transition systems*

$$\theta(\beta, \eta) =_{def} (\sigma, \eta) : T \rightarrow na(N).$$

*For a morphism of asynchronous transition systems  $(\sigma, \eta) : T \rightarrow na(N)$ , defining*

$$c\beta b \text{ iff } \emptyset \neq c = \{(s, e, s') \in Tran_* \mid b \in \sigma(s) \ \& \ b \in \sigma(s') \ \& \ b \notin \bullet\eta(e)\bullet\}$$

*yields a morphism*

$$\varphi(\sigma, \eta) =_{def} (\beta, \eta) : an(T) \rightarrow N.$$

*Furthermore,  $\theta$  and  $\varphi$  are mutual inverses, establishing a bijection between morphisms*

$$an(T) \rightarrow N$$

*and*

$$T \rightarrow na(N).$$

**Proof.** First note that  $\theta(\beta, \eta)$  and  $\varphi(\sigma, \eta)$  above are morphisms because they are the compositions

$$\theta(\beta, \eta) : T \xrightarrow{(\sigma_0, 1_E)} na \circ an(T) \xrightarrow{na(\beta, \eta)} na(N)$$

$$\varphi(\sigma, \eta) : an(T) \xrightarrow{an(\sigma, \eta)} an \circ na(N) \xrightarrow{(\beta_0, 1_{E'})} N$$

with the “unit” and “counit” morphisms of lemmas 10.2.13 and 10.2.14. We require that  $\theta, \varphi$  form a bijection.

Letting  $(\sigma, \eta) : T \rightarrow na(N)$ , we require  $\theta \circ \varphi(\sigma, \eta) = (\sigma, \eta)$ . We know  $\theta \circ \varphi(\sigma, \eta)$  has the form  $(\sigma', \eta)$ . Writing  $(\beta, \eta) =_{def} \varphi(\sigma, \eta)$  we have  $\sigma'(s) = \beta \circ M(s)$  for any  $s \in S$ . Now note that

$$\begin{aligned} b' \in \sigma'(s) &\Leftrightarrow b' \in \beta \circ M(s) \\ &\Leftrightarrow \beta^{op}(b') \in M(s) \end{aligned}$$

$$\begin{aligned} &\Leftrightarrow (s, *, s) \in \beta^{op}(b') \\ &\Leftrightarrow b' \in \sigma(s) \end{aligned}$$

where the final equivalence follows from the definition of  $\varphi$ , recalling  $(\beta, \eta) = \varphi(\sigma, \eta)$ . Thus  $\sigma' = \sigma$  and hence  $\theta \circ \varphi(\sigma, \eta) = (\sigma, \eta)$ .

To complete the proof, it is necessary to show that  $\varphi \circ \theta(\beta, \eta) = (\beta, \eta)$  for an arbitrary morphism  $(\beta, \eta) : an(T) \rightarrow N$ . Then, we write  $(\beta', \eta) =_{def} \varphi \circ \theta(\beta, \eta)$ . To show that  $\beta' = \beta$ , consider an arbitrary  $(s, e, s') \in Tran_*$ . Let  $b \in B$ . From the definitions of  $\theta$  and  $\varphi$ ,

$$(s, e, s') \in \beta'^{op}(b) \Leftrightarrow b \in \beta M(s) \ \& \ b \in \beta M(s') \ \& \ b \notin \bullet \eta(e) \bullet. \quad (\dagger)$$

Note that

$$\begin{aligned} b \in \beta M(s) &\Leftrightarrow \beta^{op}(b) \in M(s) \\ &\Leftrightarrow (s, *, s) \in \beta^{op}(b). \end{aligned}$$

Note too that, as  $(\beta, \eta)$  is a morphism,

$$b \in \bullet \eta(e) \bullet \Leftrightarrow \beta^{op}(b) \in \bullet e \bullet.$$

Hence, rewriting  $(\dagger)$ ,

$$(s, e, s') \in \beta'^{op}(b) \Leftrightarrow (s, *, s) \in \beta^{op}(b) \ \& \ (s', *, s') \in \beta^{op}(b) \ \& \ \beta^{op}(b) \notin \bullet e \bullet.$$

However, under the assumption that  $(s, *, s)$  and  $(s', *, s')$  belong to  $\beta^{op}(b)$  we have

$$\beta^{op}(b) \notin \bullet e \bullet \Leftrightarrow (s, e, s') \in \beta^{op}(b).$$

(Recall the definition of  $\bullet e$  and  $e \bullet$  in  $an(T)$ .)

Thus

$$(s, e, s') \in \beta'^{op}(b) \Leftrightarrow (s, e, s') \in \beta^{op}(b).$$

Consequently,  $\beta' = \beta$ , and we conclude that  $\varphi \circ \theta(\beta, \eta) = (\beta, \eta)$ . ■

**Theorem 10.2.16.** *The functors  $an : \mathbf{A} \rightarrow \mathbf{N}$  and  $na : \mathbf{N} \rightarrow \mathbf{A}$  form an adjunction with an left adjoint to  $na$ ; the components of the units and counits of the adjunction are the morphisms given in lemmas 10.2.13, 10.2.14.*

**Proof.** Let  $T$  be an asynchronous transition system and  $N$  a net. Let  $(\sigma_0, 1_E) : T \rightarrow na \circ an(T)$  be the morphism described in lemma 10.2.13. Let  $(\sigma, \eta) : T \rightarrow na(N)$  be a morphism in  $\mathbf{A}$ . Then, because of the bijection,  $\varphi(\sigma, \eta)$  is the unique morphism  $h : an(T) \rightarrow N$  such that

$$(\sigma, \eta) = \theta(h) = na(h) \circ (\sigma_0, 1_E)$$

—as remarked in the proof of lemma 10.2.15,  $\theta(h)$  is this composition. ■

### 10.2.2 A coreflection

Neither **A** nor **N** embeds fully and faithfully in the other category via the functors of the adjunction. This accompanies the facts that neither unit nor counit is an isomorphism (see [MacLane, 1971] p.88); in passing from a net  $N$  to  $an \circ na(N)$  extra conditions are most often introduced; the net  $an \circ na(N)$  is always safe, as we will see. While passing from an asynchronous transition system  $T$  to  $na \circ an(T)$  can not only blow up the number of states, but also collapse states which cannot be separated by conditions.

A (full) coreflection between asynchronous transition systems and nets can be obtained at the cost of adding three axioms. Let **A**<sub>0</sub> be the full subcategory of asynchronous transition systems  $T = (S, i, E, I, Tran)$  satisfying the following:

**Axiom 1** Every state is reachable from the initial state, *i.e.* for every  $s \in S$  there is a chain of events  $e_1, \dots, e_n$ , possibly empty, for which  $i \xrightarrow{e_1 \dots e_n} s$ , where  $i$  is the initial state.

**Axiom 2**  $M(u) = M(s) \Rightarrow u = s$ , for all  $s, u \in S$ .

**Axiom 3**  $*e \subseteq M(s) \Rightarrow \exists s'. (s, e, s') \in Tran$ , for all  $s \in S, e \in E$ .

Axioms 2 and 3 enforce two separation properties. The contraposition of Axiom 2 says that

$$u \neq s \Rightarrow M(u) \neq M(s),$$

*i.e.* that if two states are distinct then there is a condition of  $T$  holding at one and not the other. In fact, Axiom 2 is equivalent to

$$u \neq s \Rightarrow \exists b. b \in M(u) \ \& \ b \notin M(s)$$

though we postpone the justification of this till after we have treated *complementation* of conditions. We can recast Axiom 3 into the following form when it becomes more clearly a separation axiom. If  $(u, e, u')$  is a transition and  $s$  is a state from which there is no  $e$ -transition then there is a condition  $b$  of  $T$  such that

$$b \in M(u) \ \& \ (u, e, u') \notin b \ \& \ b \notin M(s).$$

Axioms 2 and 3 hold for any asynchronous transition system  $na(N)$  got from a net  $N$ . The proof that Axiom 3 holds uses the operation of *complementation* on conditions of an asynchronous transition system. The properties of complementation are listed below:

**Proposition 10.2.17.** *Let  $b$  be a condition of an asynchronous transition system  $T = (S, i, E, I, Tran)$ . Define*

$$\bar{b} = \{(s, e, s') \in Tran_* \mid (s, e, s') \notin b \ \& \ (s, *, s) \notin b \ \& \ (s', *, s') \notin b\}.$$

If nonempty,  $\bar{b}$  is a condition of  $T$ . It has the following properties:

$$\begin{aligned} (s, *, s) \in \bar{b} &\Leftrightarrow (s, *, s) \notin b, \text{ for any } s \in S, \\ \bar{b} \in {}^\bullet e &\Leftrightarrow b \in e^\bullet \ \& \ b \not\in {}^\bullet e \\ \bar{b} \in e^\bullet &\Leftrightarrow b \in {}^\bullet e \ \& \ b \not\in e^\bullet \text{ for any } e \in E. \end{aligned}$$

Let  $(\sigma, \eta) : T' \rightarrow T$  be a morphism of asynchronous transition systems and  $b$  be a condition of  $T$ . Then

$$(\sigma, \eta)^{-1} \bar{b} = \overline{(\sigma, \eta)^{-1} b}.$$

Suppose  $u, s$  are two distinct markings of a net  $N$ . Then certainly there is a condition  $b$  of the net in one but not the other.

Suppose for instance  $b \notin u$  and  $b \in s$ . Then, from the way the extent of a condition is defined,

$$|b| \notin M(u) \text{ and } |b| \in M(s).$$

With complementation we can separate the other way:

$$|\bar{b}| \in M(u) \text{ and } |\bar{b}| \notin M(s).$$

This justifies our earlier remark that Axiom 2 is equivalent to the seemingly stronger axiom

$$u \neq s \Rightarrow \exists b. b \in M(u) \ \& \ b \notin M(s).$$

We return to the verification that the asynchronous transition system  $na(N)$  of a net  $N$  satisfies Axioms 2 and 3.

**Proposition 10.2.18.** *Let  $N = (B, M_0, E, pre, post)$  be a net. Then  $na(N)$  satisfies the Axioms 2 and 3 above.*

**Proof.** If  $u, s$  are distinct states of  $na(N)$  they are distinct markings of  $N$  and hence only one contains some condition  $b$ . But then  $|b|$  can only be an element of one of  $M(u)$  and  $M(s)$  which are therefore unequal. This demonstrates (the contraposition of) Axiom 2.

Now we show that  $na(N)$  satisfies the contraposition of Axiom 3. Supposing  $u \xrightarrow{e} u'$  and  $s \not\xrightarrow{e}$  in  $N$ , we are required to exhibit a condition  $c$  of  $na(N)$  such that

$$c \in {}^\bullet e \ \& \ c \notin M(s).$$

There are two ways in which the marking  $s$  can fail to enable event  $e$ . Either

- (i)  $pre(e) \not\subseteq s$  or
- (ii)  $post(e) \cap (s \setminus pre(e)) \neq \emptyset$ .

In the case of (i), there is a condition  $b \in B$  of the net such that

$$b \in \text{pre}(e) \ \& \ b \notin s.$$

Hence

$$|b| \in \bullet e \ \& \ |b| \notin M(s).$$

In the case of (ii), there is a condition  $b \in B$  of the net such that

$$b \in \text{post}(e) \ \& \ b \in s \ \& \ b \notin \text{pre}(e).$$

Hence

$$|b| \in e^\bullet \ \& \ |b| \in M(s) \ \& \ |b| \notin \bullet e.$$

But then, taking the complement of  $|b|$ ,

$$\overline{|b|} \in \bullet \bar{e} \ \& \ \overline{|b|} \notin M(s),$$

by proposition 10.2.17.

In either case, (i) or (ii), we obtain a condition  $c$  of  $na(N)$  for which

$$c \in \bullet e \ \& \ c \notin M(s).$$

■

Recall that a net is *safe* if for each reachable marking  $M$  and event  $e$

$$\bullet e \subseteq M \Rightarrow e^\bullet \cap (M \setminus \bullet e) = \emptyset.$$

As we now see, if  $T$  is an asynchronous transition system which satisfies Axioms 2 and 3 then  $an(T)$  is a safe net whose behaviour is seen to be isomorphic to that of  $T$  on reachable states.

**Lemma 10.2.19.** *Assume that  $T = (S, i, E, I, \text{Tran})$  is an asynchronous transition system satisfying Axioms 2 and 3 above. Then*

1.  $e_1 I e_2 \Leftrightarrow \bullet e_1 \bullet \cap \bullet e_2 \bullet = \emptyset$  in  $an(T)$ , for any events  $e_1, e_2$ ,
2.  $(s, e, s') \in \text{Tran} \Leftrightarrow M(s) \xrightarrow{e} M(s')$  in  $an(T)$  for any  $s, s' \in S$  and  $e \in E$ ,
3.  $an(T)$  is a safe net in which every reachable marking has the form  $M(s)$  for some state  $s$  of  $T$ .

**Proof.** By lemma 10.2.10,

$$\begin{aligned} e_1 I e_2 &\Leftrightarrow \bullet e_1 \bullet \cap \bullet e_2 \bullet = \emptyset, \\ (s, e, s') \in \text{Tran} &\Rightarrow M(s) \xrightarrow{e} M(s') \text{ in } an(T). \end{aligned}$$

This yields (1) and (2)“ $\Rightarrow$ ”. To establish the converse, (2)“ $\Leftarrow$ ”, with the assumption of Axioms 2 and 3, suppose  $M(s) \xrightarrow{e} M(s')$ . Then  $\bullet e \subseteq M(s)$



so  $(s, e, s_1) \in \text{Tran}$  from some state  $s_1$  by Axiom 3. Thus  $M(s) \xrightarrow{e} M(s_1)$  and so  $M(s') = M(s_1)$ . Now by Axiom 2 we deduce  $s' = s_1$ , and hence the converse

$$M(s) \xrightarrow{e} M(s') \Rightarrow (s, e, s') \in \text{Tran}.$$

We now show (3). Any reachable marking of  $\text{an}(T)$  has the form  $M(s)$  for some  $s \in S$  by the following argument. Assuming  $M(s) \xrightarrow{e} M_1$  we necessarily have  $\bullet e \subseteq M(s)$  whereupon, as above, there is a transition  $(s, e, s_1)$  of  $T$  with  $M_1 = M(s_1)$ ; thus, by induction along any reachability chain, any reachable marking of  $\text{an}(T)$  is of the form  $M(s)$  for some state  $s$  of  $T$ . Because the two sets

$$\begin{aligned} e^\bullet &= \{b \in M(s') \mid (s, e, s') \notin b\}, \\ M(s) \setminus \bullet e &= \{b \in M(s) \mid (s, e, s') \in b\} \end{aligned}$$

are clearly disjoint, the net  $\text{an}(T)$  is safe. ■

**Corollary 10.2.20.** *For any net  $N$ , the net  $\text{an} \circ \text{na}(N)$  is safe.*

The coreflection between  $\mathbf{A}_0$  and  $\mathbf{N}$  is defined using a simple coreflection between the full subcategory of  $\mathbf{A}$ , consisting of objects, where all states are reachable, and  $\mathbf{A}$ .

**Definition 10.2.21.** Let  $\mathbf{A}^R$  be the full subcategory of  $\mathbf{A}$  consisting of asynchronous transition systems  $(S, i, E, I, \text{Tran})$  satisfying Axiom 1, i.e. so that all states  $s$  are reachable.

Let  $\mathcal{R}$  act on an asynchronous transition system  $T = (S, i, E, I, \text{Tran})$  as follows:

$$\mathcal{R}(T) = (S', i', E', I', \text{Tran}')$$

where

$$\begin{aligned} S' & \text{ consists of all reachable states of } T \\ E' &= \{e \in E \mid \exists s, s' \in S'. (s, e, s') \in \text{Tran}\} \\ I' &= I \cap (E' \times E') \\ \text{Tran}' &= \text{Tran} \cap (S' \times E' \times S'). \end{aligned}$$

For a morphism  $(\sigma, \eta) : T \rightarrow T'$  of asynchronous transition systems, define  $\mathcal{R}(\sigma, \eta) = (\sigma', \eta')$  where  $\sigma'$  and  $\eta'$  are the restrictions of  $\sigma$  and  $\eta$  to the states, respectively events, of  $\mathcal{R}(T)$ .

We note that a morphism from an asynchronous transition system in which all states are reachable is determined by how it acts on events:

**Proposition 10.2.22.** *Suppose  $(\sigma, \eta)$  and  $(\sigma', \eta)$  are morphisms  $T \rightarrow T'$  between asynchronous systems where each state of  $T$  is reachable. Then  $\sigma = \sigma'$ .*

**Proof.** An obvious consequence of the determinacy property

$$(s, e, s_1) \in \text{Tran} \ \& \ (s, e, s_2) \in \text{Tran} \Rightarrow s_1 = s_2$$

of asynchronous transition systems. ■

**Proposition 10.2.23.** *The operation  $\mathcal{R}$  is a functor  $\mathbf{A} \rightarrow \mathbf{A}^R$  which is right adjoint to the inclusion functor  $\mathcal{I} : \mathbf{A}^R \rightarrow \mathbf{A}$ . The unit of the adjunction at  $T \in \mathbf{A}^R$  is the identity on  $T$ , making the adjunction a coreflection. The counit at  $T \in \mathbf{A}^R$  is given by  $(j_S, j_E) : \mathcal{R}(T) \rightarrow T$  where  $j_S$  and  $j_E$  are the inclusion maps on states and events respectively. Moreover,  $\mathcal{R}$  preserves Axioms 2 and 3 in the sense that if  $T$  satisfies Axiom 2 (or 3) then  $\mathcal{R}(T)$  satisfies Axiom 2 (or 3).*

**Proof.** We omit the straightforward proof that  $\mathcal{R}$  is a right adjoint to the inclusion of categories with counit as claimed. Let  $j : \mathcal{R}(T) \rightarrow T$  be a component of the counit. The transitions  $\text{Tran}'$  of  $\mathcal{R}(T)$  are a subset of those of  $T$ . If  $b$  is a condition of  $T$  then  $j^{-1}b = b \cap \text{Tran}'$  is a condition of  $\mathcal{R}(T)$  provided it is nonempty. Suppose  $s_1$  and  $s_2$  are two distinct states of  $\mathcal{R}(T)$ . If  $T$  satisfies Axiom 2 then there is a condition  $b$  of  $T$  such that one and only one of  $(s_1, *, s_1) \in b$ ,  $(s_2, *, s_2) \in b$  holds. But then  $j^{-1}b$  is a condition of  $\mathcal{R}(T)$  separating  $s_1, s_2$ . Thus  $\mathcal{R}$  preserves Axiom 2, and by a similar argument, Axiom 3. ■

We show the adjunction, with  $an$  left adjoint to  $\mathcal{R} \circ na$ , obtained as the composition forms a coreflection. Its counit is given by the notion of *reachable extent* of a condition. This consists essentially of the reachable markings and transitions at which  $b$  holds uninterruptedly.

**Definition 10.2.24.** Let  $N$  be a net. Let  $\text{Tran}_*$  be the transitions and idle transitions of  $\mathcal{R} \circ na(N)$ . Define

$$|b|^R = |b| \cap \text{Tran}_*.$$

**Theorem 10.2.25.** *Defining  $na_0 = \mathcal{R} \circ na$ , the composition of functors, yields a functor  $na_0 : \mathbf{N} \rightarrow \mathbf{A}_0$  which is right adjoint to  $an_0 : \mathbf{A}_0 \rightarrow \mathbf{N}$ , the restriction of  $an$  to  $\mathbf{A}_0$ .*

*The unit at  $T = (S, i, E, I, \text{Tran}) \in \mathbf{A}_0$  is an isomorphism*

$$(\sigma, 1_E) : T \rightarrow na_0 \circ an(T)$$

*where  $\sigma(s) = M(s)$  for  $s \in S$ , making the adjunction a coreflection.*

*The counit at a net  $N$  is*

$$(\beta, 1_E) : an \circ na_0 \rightarrow N$$

*where*

$$c\beta b \text{ iff } \emptyset \neq c = |b|^R$$

between conditions  $c$  of  $na_0(N)$  and  $b$  of  $N$

**Proof.** The adjunctions compose to give  $\mathcal{R} \circ na : \mathbf{N} \rightarrow \mathbf{A}^R$  a right adjoint to  $\mathcal{I} \circ an : \mathbf{A}^R \rightarrow \mathbf{N}$ . However, the image  $\mathcal{R} \circ na(N)$  of a net  $N$  always satisfies Axioms 2 and 3 as well as 1. This is because  $na(N)$  satisfies Axioms 2 and 3, and  $\mathcal{R}$  preserves these axioms. Thus the adjunction cuts down to one where  $na_0 : \mathbf{N} \rightarrow \mathbf{A}_0$  is right adjoint to  $an_0 : \mathbf{A}_0 \rightarrow \mathbf{N}$ . It is an adjunction with unit at  $T = (S, i, E, I, Tran) \in \mathbf{A}_0$  a morphism in  $\mathbf{A}_0$

$$(\sigma, 1_E) : T \rightarrow na_0 \circ an(T)$$

where  $\sigma(s) = M(s)$  for  $s \in S$ .

That the unit  $(\sigma, 1_E) : T \rightarrow na_0 \circ an(T)$  is an isomorphism follows from lemma 10.2.19. Hence the functors  $an$ ,  $na_0$  form a coreflection with  $an_0$  left adjoint to  $na_0$ .

That the counit has the form claimed follows by composing the natural bijections of the adjunctions given by proposition 10.2.23 and lemma 10.2.15. ■

One consequence of the coreflection is that any net  $N$  can be converted to a safe net  $an \circ na_0(N)$  with the same behaviour, in the sense that that there is an isomorphism between the reachable asynchronous transition systems the two nets induce under  $na_0$ . Another is that  $\mathbf{A}_0$  has products and coproducts given by the same constructions as those of  $\mathbf{A}$ .

**Proposition 10.2.26.** *The category  $\mathbf{A}_0$  has products and coproducts which coincide with those in the category  $\mathbf{A}$ .*

**Proof.** The product of nets in  $\mathbf{N}$  becomes the product in  $\mathbf{A}_0$  of asynchronous transition systems under  $na_0$ . Its behaviour, which is described in proposition 9.2.7, ensures that its image under  $na_0$  coincides with the product in  $\mathbf{A}$ .

The coproduct in  $\mathbf{A}$  will be the coproduct in  $\mathbf{A}_0$  provided it is the image to within isomorphism of a net. However, if  $T_0, T_1$  are objects of  $\mathbf{A}_0$ , then by lemma 9.2.4 their coproduct in  $\mathbf{A}$  is isomorphic to  $na_0(an(T_0) + an(T_1))$ . ■

The coreflection  $\mathbf{A}_0 \hookrightarrow \mathbf{N}$  cuts down to an equivalence of categories by restricting to the appropriate full subcategory of nets.

**Definition 10.2.27.** Let  $\mathbf{N}_0$  be the full subcategory on nets such that

$$b \mapsto |b|^R$$

is a bijection between conditions of  $N$  and those of  $na_0(N)$ .

**Theorem 10.2.28.** *The functor  $an$  restricts to a functor  $an_0 : \mathbf{A}_0 \rightarrow \mathbf{N}_0$ . The functor  $\mathcal{R} \circ na$  restricts to a functor  $na_0 : \mathbf{N}_0 \rightarrow \mathbf{A}_0$ . The functors  $an_0, na_0$  form an equivalence of categories.*

**Proof.** Recall the coreflection of theorem 10.2.25:  $na_0 = \mathcal{R} \circ na : \mathbf{N} \rightarrow \mathbf{A}_0$  is right adjoint to  $an_0 : \mathbf{A}_0 \rightarrow \mathbf{N}$ , the restriction of  $an$  to  $\mathbf{A}_0$ . The counit of the coreflection, at a net  $N$ ,

$$(\beta, 1_E) : an_0 \circ na_0(N) \rightarrow N$$

has  $c\beta b$  iff  $c = |b|^R$ , between conditions. This is an isomorphism iff  $N \in \mathbf{N}_0$ . We thus obtain an equivalence of categories. ■

Nets in  $\mathbf{N}_0$  are saturated with conditions in the sense that they have as many conditions as is allowed by their reachable behaviour and independence (regarded as an asynchronous transition system). Nets in  $\mathbf{N}_0$  cannot, however, have more than one copy of a condition with particular starting and ending events (they are *condition-extensional*). This is because:

**Proposition 10.2.29.** *Let  $T$  be an asynchronous transition system for which each state is reachable. If  $b_1, b_2$  are conditions of  $T$  for which*

$$\bullet b_1 = \bullet b_2 \quad \text{and} \quad b_1^\bullet = b_2^\bullet$$

*then*

$$b_1 = b_2.$$

**Proof.** Suppose  $\bullet b_1 = \bullet b_2$  and  $b_1^\bullet = b_2^\bullet$  for conditions  $b_1, b_2$  of  $T$ . Inductively along a chain of transitions

$$(i, e_1, s_1), (s_1, e_2, s_2), \dots, (s_{n-1}, e_n, s_n)$$

the membership of  $(s_{i-1}, e_i, s_i)$  (or  $(s_i, *, s_i)$ ) in  $b_1$  and in  $b_2$  must agree. ■

If on the other hand an asynchronous transition system  $T$  has a state which is not reachable then there will be distinct conditions of  $T$  with the same end points. Suppose  $T$  has states which are not reachable and let  $Tran_0$  be all transitions, including idle ones, which are not reachable. If  $b_1$  is a condition, say consisting solely of reachable transitions of  $T$ , then so is  $b_2 = b_1 \cup Tran_0$  a condition, necessarily distinct from  $b_1$ , but with  $\bullet b_1 = \bullet b_2$  and  $b_1^\bullet = b_2^\bullet$ .

We have already observed the coreflection from event structures to asynchronous transition systems  $\mathbf{E} \hookrightarrow \mathbf{A}$ . In fact the coreflection cuts down to one between  $\mathbf{E} \hookrightarrow \mathbf{A}_0$ .

**Proposition 10.2.30.** *For any event structure  $E$ , the asynchronous transition system  $tla \circ etl(E)$  is an object in  $\mathbf{A}_0$ . Consequently,  $tla \circ etl$  cuts*

down to a functor  $\mathbf{E} \rightarrow \mathbf{A}_0$  left adjoint to the restriction of  $atl \circ tle$  to  $\mathbf{A}_0 \rightarrow \mathbf{E}$  also forming a coreflection.

**Proof.** The functor  $tla \circ etl : \mathbf{E} \rightarrow \mathbf{A}$  is left adjoint to  $atl \circ tle : \mathbf{A} \rightarrow \mathbf{E}$  and forms a coreflection. It suffices to show that  $tla \circ etl(E)$  is an object of  $\mathbf{A}$  for any event structure  $E$ . Let  $E$  be an event structure. Note that  $tla \circ etl(E)$  is an asynchronous transition system isomorphic to the asynchronous transition system with transitions  $x \xrightarrow{e} x'$ , between finite configurations of  $E$ , and independence relation  $co$ —see proposition 10.1.7. There are many ways of adjoining conditions to events of an event structure so as to produce a (safe) net  $N$  with reachable transition and independence relations isomorphic to that of the configurations of  $E$  (see *e.g.* the construction of an occurrence net from an event structure in [Nielsen *et al.*, 1981]). Hence by theorem 10.2.25,  $na_0(N)$ , and so the isomorphic  $tla \circ etl(E)$ , belong as objects to  $\mathbf{A}_0$ . ■

### 10.3 Properties of conditions

In this section we explore briefly the “logical” properties of conditions of an asynchronous transition system. These follow from the construction of conditions out of special subsets. A general condition can be decomposed into a disjoint union of minimal components, called connected conditions. As will be seen, the behaviour of the net constructed from an asynchronous transition system is determined by just its connected conditions. The notion of connected condition appears automatically in establishing an adjunction between asynchronous transition systems and a previously studied category of safe nets. This category of nets has a broader class of morphisms, ones which allow general foldings.

Let us first summarise the properties of conditions of an asynchronous transition system that have arisen in the proofs above:

- If  $b$  is a condition of an asynchronous transition system  $T$  and  $(\sigma, \eta) : T' \rightarrow T$  is a morphism of asynchronous transition systems, then the inverse image  $(\sigma, \eta)^{-1}b$  is a condition of  $T'$ , if nonempty (*cf.* lemma 10.2.7).
- If  $X$  is a nonempty pairwise-dependent subset of events of an asynchronous transition system  $T$  (so  $e_1 I e_2$  for all  $e_1, e_2 \in X$ ) then there is a condition of  $T$  started by precisely the events  $X$  and ended by precisely  $X$  (*cf.* lemma 10.2.9).
- Any condition  $b$  of an asynchronous transition system has a complement  $\bar{b}$ , either empty or a condition whose starting and ending events reverse those of  $b$  (*cf.* proposition 10.2.17).

In general, conditions of an asynchronous transition system are not closed under intersections and unions. However, in the case where the unions are disjoint:



- If  $S$  is a nonempty collection of conditions of an asynchronous transition system  $T$  which is disjoint, in the sense that

$$\text{if } b_1 \cap b_2 \neq \emptyset \text{ then } b_1 = b_2, \text{ for all } b_1, b_2 \in S,$$

then  $\bigcup S$  is a condition of  $T$ .

### 10.3.1 Connected conditions

A new condition can be built up out of disjoint components. And in fact, conversely, any condition can be decomposed into a disjoint union of *connected* components which cannot be decomposed further.

**Definition 10.3.1.** Let  $T$  be an asynchronous transition system with conditions  $B$ . For  $b, b' \in B \cup \{\emptyset\}$ , write

$$b' \text{ comp } b \Leftrightarrow_{def} b' \subseteq b \text{ \& } b \setminus b' \in B \cup \{\emptyset\}.$$

(If  $b' \text{ comp } b$  we say  $b'$  is a *component* of  $b$ .)

Say a condition  $b$  is *connected* iff there are *not* conditions  $b_1, b_2$  such that

$$b = b_1 \cup b_2 \text{ \& } b_1 \cap b_2 = \emptyset.$$

**Notation 10.3.2.** We shall write  $b = b_1 \dot{\cup} b_2$  to mean  $b = b_1 \cup b_2$  and  $b_1 \cap b_2 = \emptyset$ , for sets  $b, b_1, b_2$ .

In fact, a component of a condition  $b$  of an asynchronous transition system is either empty or a condition included in  $b$  whose boundary of starting/ending events agrees with that of  $b$ :

**Lemma 10.3.3.** Let  $c, b$  be conditions of an asynchronous transition system  $T$ . Then,

$$\begin{aligned} c \text{ comp } b &\Leftrightarrow c \subseteq b \text{ \& for all transitions } (s, e, s') \text{ of } T \\ &\quad (s, e, s') \in {}^\bullet c \Rightarrow (s, e, s') \in {}^\bullet b \text{ \& } \\ &\quad (s, e, s') \in c^\bullet \Rightarrow (s, e, s') \in b^\bullet. \end{aligned}$$

**Proof.** By basic set theory from the definition of a condition of an asynchronous transition system. ■

**Proposition 10.3.4.** Let  $T$  be an asynchronous transition system with conditions  $B$ .

- (i) The relation *comp* is a partial order on  $B \cup \{\emptyset\}$ .
- (ii) A condition is *connected* iff it is a minimal condition with respect to *comp*.
- (iii) If  $c_i, i \in I$ , is a family of components of a condition  $b$ , then

$$\bigcap_{i \in I} c_i, \quad \bigcup_{i \in I} c_i$$

are also components of  $b$ .

(iv) If  $c_1$  and  $c_2$  are components of a condition  $b$ , then  $c_1 \setminus c_2$  is also a component of  $b$ .

**Proof.**

(i) Clearly comp is reflexive. Suppose  $c \text{ comp } b \text{ comp } a$ . Then  $c \subseteq b$  and  $b \setminus c$  is a condition if nonempty as well as  $b \subseteq a$  and  $a \setminus b$  forming a condition if it is nonempty. It follows that  $c \subseteq a$  and that

$$a \setminus c = (a \setminus b) \dot{\cup} (b \setminus c)$$

and hence  $a \setminus c$  is itself a condition if nonempty. But this makes  $c \text{ comp } a$ .  
(ii) Clear.

(iii) Assume  $c_i, i \in I$ , is a family of components of a condition  $b$ . It can be checked that  $\bigcup_{i \in I} c_i$  and  $\bigcap_{i \in I} c_i$  satisfy the axioms required of conditions of  $T$ , if nonempty.

To give the flavour of the arguments, assume  $(s, e, s') \in (\bigcup_i c_i)^\bullet$  and that  $(u, e, u')$  is a transition of which we wish to show  $(u, e, u') \in (\bigcup_i c_i)^\bullet$ —one of the properties needed for the union to be a condition. As  $(s, e, s') \in (\bigcup_i c_i)^\bullet$ , there is a component  $c_j$  for which  $(s, e, s') \in c_j^\bullet$ . But then  $(s, e, s') \in b^\bullet$  (lemma 10.3.3). As  $b$  is a condition,  $(u, e, u') \in b^\bullet$  and, in particular,  $(u, e, u') \notin b$ . As  $c_j$  is a condition,  $(u, e, u') \in c_j^\bullet$ , and in particular  $(u, *, u) \in c_j$ . But then  $(u, *, u) \in \bigcup_i c_i$ , and  $(u, e, u') \notin \bigcup_i c_i$  as  $\bigcup_i c_i \subseteq b$ . This makes  $(u, e, u') \in (\bigcup_i c_i)^\bullet$ .

The conditions  $\bigcup_{i \in I} c_i$  and  $\bigcap_{i \in I} c_i$  are also components because the complements

$$\begin{aligned} b \setminus \bigcup_i c_i &= \bigcap_i (b \setminus c_i) \\ b \setminus \bigcap_i c_i &= \bigcup_i (b \setminus c_i) \end{aligned}$$

and, as we have just remarked, intersections and unions of components of  $b$  form conditions if nonempty.

(iv) Finally, suppose  $c_1$  and  $c_2$  are components of  $b$ . Then  $(b \setminus c_1) \cup c_2$  is a union of components of  $b$ , and hence itself a component, ensuring that  $b \setminus ((b \setminus c_1) \cup c_2)$  is a component of  $b$ . Note that  $c_1 \setminus c_2 = b \setminus ((b \setminus c_1) \cup c_2)$ . ■

**Definition 10.3.5.** Let  $b$  be a condition of an asynchronous transition system. Define

$$\text{conn}(b) = \{c \mid c \text{ is a connected condition \& } c \text{ comp } b\}.$$

The family of connected components  $\text{conn}(b)$ , of a condition  $b$  is disjoint and covers  $b$  in the following sense:

**Lemma 10.3.6.** *Given a condition  $b$  of an asynchronous transition system,*

- (i)  $c_1 \in \text{conn}(b)$  &  $c_2 \in \text{conn}(b)$  &  $c_1 \cap c_2 \neq \emptyset \Rightarrow c_1 = c_2$ , and
- (ii)  $b = \bigcup \text{conn}(b)$ .

**Proof.**

(i) Suppose  $c_1, c_2 \in \text{conn}(b)$  and  $c_1 \cap c_2 \neq \emptyset$ . It follows that  $c_1 \cap c_2$  and  $c_1 \setminus c_2$  are components of  $b$  if nonempty, with

$$c_1 = (c_1 \setminus c_2) \dot{\cup} (c_1 \cap c_2).$$

But  $c_1$  is connected so  $c_1 \setminus c_2 = \emptyset$  yielding  $c_1 \subseteq c_2$ . Similarly,  $c_2 \subseteq c_1$ , implying  $c_1 = c_2$ .

(ii) Clearly  $\bigcup \text{conn}(b) \subseteq b$ . To show the converse inclusion, let  $t \in b$ . Take

$$d = \bigcap \{c \mid t \in c \text{ \& } c \text{ comp } b\}.$$

Being an intersection of  $b$ 's components,  $d$  is itself a component of  $b$ . Clearly it contains  $t$ , is a minimal component of  $b$ , and so is connected. Thus  $t \in d$  and  $d \in \text{conn}(b)$ , ensuring  $t \in \bigcup \text{conn}(b)$ . ■

Recall the construction  $an(T)$  of a net from an asynchronous transition system  $T$ . When  $T$  is an object of  $\mathbf{A}_0$ , the net  $an(T)$  is safe and has behaviour isomorphic to that of  $T$  (cf. lemma 10.2.19). In fact we can restrict the construction to just the connected conditions—these are sufficient to determine the net's behaviour.

**Definition 10.3.7.** Let  $T$  be an asynchronous transition system in  $\mathbf{A}_0$ . Assume  $an(T)$  has the form  $(B, M_0, E, pre, post)$ . Define  $an_c(T) = (C, M_0 \cap C, E, pre_c, post_c)$  where  $C$  consists of the connected conditions of  $T$  and

$$pre_c(e) = pre(e) \cap C, \quad post_c(e) = post(e) \cap C.$$

**Lemma 10.3.8.** *Let  $T$  be an asynchronous transition system in  $\mathbf{A}_0$ . Then:*

- (i)  $an_c(T)$  is a safe net.
- (ii) There is an isomorphism  $(\sigma, 1_E) : T \rightarrow na_0 \circ an_c(T)$  where  $\sigma(s) = M(s) \cap C$  for  $s$  a state of  $T$  with connected conditions  $C$ .

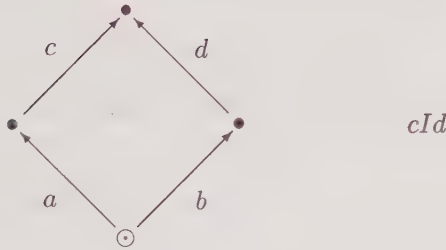
**Proof.** The separation axioms, Axiom 2 and Axiom 3, hold iff they hold restricted to only connected conditions. For this reason the proof proceeds pretty much as that of lemma 10.2.19. There are a couple of refinements. First showing that  $an_c(T)$  is a net requires that  $pre_c(e)$  and  $post_c(e)$

are nonempty for any event  $e$ . As earlier, in the proof of lemma 10.2.10, lemma 10.2.9 yields  $b$  as a pre- and post-condition of  $e$ , though it is not necessarily connected; however, amongst its connected components  $\text{conn}(b)$  are connected pre- and post-conditions of  $e$ . An extra argument is needed in showing that independence of the net  $an_c(T)$  coincides with that of  $T$ , as is required for the isomorphism of (ii). This involves showing that if  $\neg e_1 I e_2$  in  $T$  then there is a *connected* condition  $c$  in  $\bullet e_1 \bullet \cap \bullet e_2 \bullet$ . By lemma 10.2.9, there is a condition  $b$  (not necessarily connected) such that

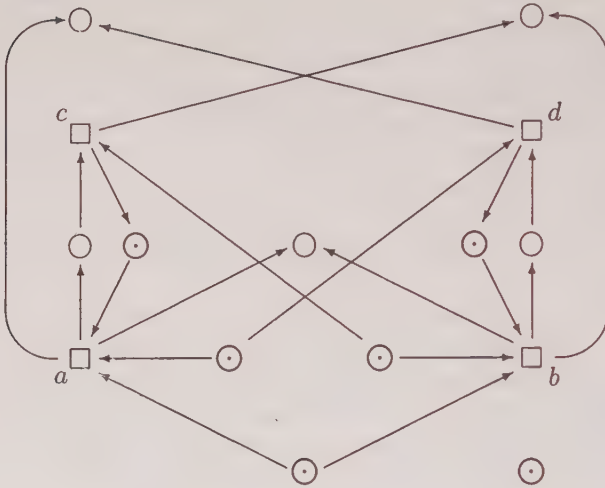
$$\{e_1, e_2\} = \{e \mid b \in \bullet e\} = \{e \mid b \in e \bullet\}.$$

As  $T$  is an asynchronous transition system in which all transitions are reachable, it has transitions  $(s_1, e_1, s'_1)$  and  $(s_2, e_2, s'_2)$  that are connected by a chain of transitions (backwards and then forwards from the initial state) which do *not* involve the events  $e_1, e_2$ . This chain must be included in a single connected component  $c$  of  $b$ —otherwise, by lemma 10.3.3,  $c$  and so  $b$  would be started or ended by an event other than  $e_1$  or  $e_2$ . This connected component is one for which  $c \in \bullet e_1 \bullet \cap \bullet e_2 \bullet$ . ■

**Example 10.3.9.** The asynchronous transition system



gives rise, under  $an_c$ , to the following net, in which every condition is connected:



### 10.3.2 Relational morphisms on nets

With the aid of connected conditions we can link to another definition of morphism on nets. A limitation with the definition of morphism on nets (in section 9.1) is that it does not permit “folding” morphisms of the kind illustrated in the example below.

Let us temporarily broaden our attention to Petri nets in which conditions can hold with multiplicities, in other words to general Petri nets, though without capacities. Typically such a net consists of

$$(B, M_0, E, pre, post)$$

with much the same intuition as before, but where the initial marking  $M_0$  is now a *multiset* of the set of conditions  $B$  and  $pre$  and  $post$  are now *multirelations* specifying the multiplicity of conditions used or produced by an event in  $E$ . Letting  $M, M'$  be markings (*i.e.* multisets of conditions) and  $A$  be a finite multiset of events we define  $M \xrightarrow{A} M'$  iff

$$pre.A \leq M \text{ and } M' = M - (pre.A) + (post.A).$$

Here we are using a little multiset notation. Thinking of multisets as vectors (of possibly infinite dimension) and multirelations as matrices (possibly infinite), we can compare two multisets coordinatewise with the relation  $\leq$ , add, and subtract multisets (provided no component goes negative).



Using matrix multiplication we can apply a multirelation to a multiset as in *pre.A* which yields the multiset of preconditions of the multiset of events *A*. The fact that *A* is a multiset reveals concurrency amongst event occurrences; for example, a transition

$$M \xrightarrow{3e+2f} M'$$

is interpreted as meaning that three occurrences of the event *e* occur concurrently with each other and with two of *f* in taking the marking *M* to *M'*.<sup>9</sup>

What about morphisms between two general Petri nets  $N = (B, M_0, E, \text{pre}, \text{post})$ ,  $N' = (B', M'_0, E', \text{pre}', \text{post}')$ ? Taking account of multiplicities, and at the same time respecting events, it is reasonable to take a morphism to be  $(\beta, \eta) : N \rightarrow N'$  where  $\beta$  is a multirelation from *B* to *B'* and  $\eta$  is a partial function from *E* to *E'* (which in this context will be understood as a special kind of multirelation) such that

$$\begin{aligned} \beta.M_0 &= M'_0 \\ \beta(\text{pre}.e) &= \text{pre}' . (\eta.e) \text{ and} \\ \beta(\text{post}.e) &= \text{post}' . (\eta.e), \text{ for all } e \in E. \end{aligned}$$

Such morphisms preserve behaviour:

**Proposition 10.3.10.** *Let  $(\beta, \eta) : N \rightarrow N'$  be a morphism of general nets. If *M* is a reachable marking of *N* and  $M \xrightarrow{A} M'$  in *N*, then  $\beta.M$  is a reachable marking of *N'* and  $\beta.M \xrightarrow{\eta \cdot A} \beta.M'$  in *N'*.*

**Proof.** A straightforward induction on the number of steps to the reachable marking *M*. ■

Provided we restrict attention to general Petri nets in which every condition occurs with nonzero multiplicity in either the initial marking or the pre- or postconditions of some event, we can form a category by taking the composition of

$$(\beta, \eta) : N \rightarrow N' \text{ and } (\beta', \eta') : N' \rightarrow N''$$

to be  $(\beta' \circ \beta, \eta' \circ \eta)$  where  $\beta' \circ \beta$  is the multirelation composition of  $\beta'$  and  $\beta$ , and  $\eta' \circ \eta$  is a composition of partial functions—without the restriction the composition could yield infinite multiplicities.

This whole set-up makes sense for safe nets, and we can define **Safe** to be the full subcategory of general Petri nets in which objects are safe

---

<sup>9</sup>A thorough treatment of multisets and multirelations can be found in the appendix of [Winskel, 1987b].

nets for which every condition belongs to either the initial marking or the pre- or postcondition of some event. For such safe nets if  $M$  is a reachable marking and  $M \xrightarrow{A} M'$  then the multiset of events  $A$  has no multiplicity exceeding 1, *i.e.*  $A$  can be identified with a subset of events, which are to be thought of as occurring concurrently. For later, we define the concurrency notation on events  $e_1, e_2$  by taking

$$e_1 \text{ co } e_2 \Leftrightarrow_{\text{def}} M \xrightarrow{\{e_1, e_2\}} M' \text{ for some reachable markings } M, M'.$$

In a safe net the relation  $e_1 \text{ co } e_2$  amounts to there existing a reachable marking  $M$  for which

$$\bullet e_1 \subseteq M \quad \& \quad \bullet e_2 \subseteq M \quad \& \quad \bullet e_1 \cap \bullet e_2 = \emptyset,$$

or equivalently, the two events  $e_1, e_2$  are independent and both have concession at some reachable marking.

A little work shows that morphisms in **Safe**, between  $N = (B, M_0, E, \text{pre}, \text{post})$ ,  $N' = (B', M'_0, E', \text{pre}', \text{post}')$ , can be given equivalently as  $(\beta, \eta) : N \rightarrow N'$  where  $\beta \subseteq B \times B'$  is a relation and  $\eta : E \rightarrow_* E'$  such that

$$\beta M_0 \subseteq M'_0 \text{ and } \forall b' \in M'_0 \exists! b \in M_0. b\beta b',$$

and for any event  $e \in E$ ,

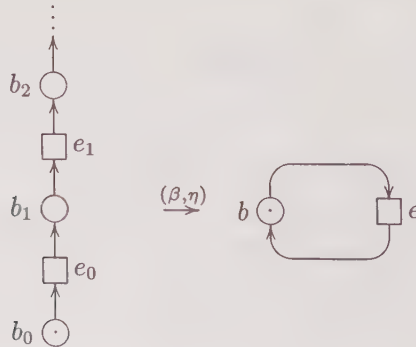
$$\beta \text{pre}(e) \subseteq \text{pre}'(\eta(e)) \text{ and } \forall b' \in \text{pre}'(\eta(e)) \exists! b \in \text{pre}(e). b\beta b',$$

and

$$\beta \text{post}(e) \subseteq \text{post}'(\eta(e)) \text{ and } \forall b' \in \text{post}'(\eta(e)) \exists! b \in \text{post}(e). b\beta b',$$

where the character of multiset application reappears in the form of uniqueness restrictions local to the initial marking and neighbourhoods of events. These say that  $\beta^{op}$  is a function local to  $M'_0$ ,  $\text{pre}'(\eta(e))$ , and  $\text{post}'(\eta(e))$ . It is not required that  $\beta^{op}$  be a partial function globally. The added generality permits the following kind of morphism:

**Example 10.3.11.** Consider the “folding” morphism in **Safe**



sending each event  $e_0, e_1, \dots$  to the common event  $e$ , and each condition  $b_0, b_1, \dots$  to the condition  $b$ , so

$$\eta(e_i) = e \text{ and } b_i \beta b$$

for  $i \in \omega$ . Note that while this morphism does preserve the concurrency relation  $co$  it does not preserve independence and, for example, we have  $\bullet e_1 \cap \bullet e_3 = \emptyset$ , making the events  $e_1$  and  $e_3$  independent whereas their common image  $e$  cannot be independent with itself.

So morphisms in **Safe** do not preserve the independence relation on nets if we interpret independence as disjointness of pre- and post-conditions. But the morphisms do preserve the concurrency relation. We can obtain a functor from **Safe** to asynchronous transition systems by instead interpreting independence as the concurrency relation on safe nets. The functor will map into the category  $\mathbf{A}_c$ —the full subcategory of  $\mathbf{A}_0$  consisting of objects for which the following property holds of the independence relation  $I$ :

$$e_1 I e_2 \Rightarrow s \xrightarrow{e_1} s_1 \quad \& \quad s \xrightarrow{e_2} s_2 \text{ for some states } s, s_1, s_2.$$

**Proposition 10.3.12.**  $\mathbf{A}_c$  is a coreflective subcategory of  $\mathbf{A}_0$ .

**Proof.** The inclusion functor has a right adjoint  $V$  which from an object  $T = (S, i, E, I, Tran)$  in  $\mathbf{A}_0$  produces  $V(T) = (S, i, E, I', Tran)$  in which the independence relation is restricted so

$$e_1 I' e_2 \Leftrightarrow_{def} e_1 I e_2 \quad \& \quad s \xrightarrow{e_1} s_1 \quad \& \quad s \xrightarrow{e_2} s_2 \text{ for some states } s, s_1, s_2.$$

■

We use  $V$ , the right adjoint of the coreflection  $\mathbf{A}_c \hookrightarrow \mathbf{A}_0$ , to obtain a functor from **Safe** to  $\mathbf{A}_c$ . For  $N$  in **Safe**, define

$$na_c(N) = V \circ na_0(N),$$

an asynchronous transition system whose states are the reachable markings of  $N$  and with independence the concurrency relation on  $N$ . For a morphism,  $(\beta, \eta) : N \rightarrow N'$  in **Safe**, define  $na_c(\beta, \eta) = (\sigma, \eta)$  where  $\sigma(M) = \beta M$  for any reachable marking  $M$  of  $N$ .

Via the next lemma, the coreflection between  $\mathbf{A}_0 \hookrightarrow \mathbf{N}$  yields a coreflection  $\mathbf{A}_c \hookrightarrow \mathbf{Safe}$ .

**Lemma 10.3.13.** Let  $T$  be an asynchronous transition system in  $\mathbf{A}_0$  with events  $E$ .

(i) There is a morphism in **Safe**

$$(\gamma, 1_E) : an_c(T) \rightarrow an_0(T)$$

with  $c\gamma b \Leftrightarrow c \in \text{conn}(b)$ .

- (ii) For any morphism  $(\beta, \eta) : an_c(T) \rightarrow N$  in **Safe**, there is a unique morphism  $(\varphi, \eta) : an_0(T) \rightarrow N$  in **N** such that the following diagram commutes:

$$\begin{array}{ccc} an_c(T) & \xrightarrow{(\gamma, 1_E)} & an_0(T) \\ & \searrow (\beta, \eta) & \downarrow (\varphi, \eta) \\ & & N \end{array}$$

**Proof.** Assume  $T$  is an asynchronous transition system and that

$$\begin{aligned} an_0(T) &= (B, M_0, E, pre, post), \\ an_c(T) &= (C, M_0 \cap C, E, pre_c, post_c) \end{aligned}$$

where  $C$  consists of the connected conditions of  $T$ , and

$$pre_c(e) = pre(e) \cap C, \quad post_c(e) = post(e) \cap C$$

for any event  $e$ . For a state  $s$  of  $T$ , we write  $M_c(s) = M(s) \cap C$ . In particular,  $M_0 \cap C = M_c(i)$ , where  $i$  is the initial state of  $T$ .

- (i) We show that  $(\gamma, 1_E)$  is a morphism in **Safe**, according to the definition of this section.

Let  $c \in M_c(i)$ , where  $i$  is the initial state of  $T$ . If  $c\gamma b$  then  $(i, *, i) \in c$ , so  $(i, *, i) \in b$  giving  $b \in M(i)$ . Thus  $\gamma M_0 \cap C \subseteq M_0$ . Suppose  $c_1, c_2 \in M_c(i)$  with  $c_1\gamma b$  and  $c_2\gamma b$ . Then  $(i, *, i) \in c_1 \cap c_2$  where  $c_1, c_2 \in \text{conn}(b)$ . By lemma 10.3.6,  $c_1 = c_2$ .

Suppose  $c \in pre_c(e)$ , for  $e \in E$ , and  $c\gamma b$ . Then there is a transition  $(s, e, s')$  such that  $(s, e, s') \in c^\bullet$ . By lemma 10.3.3,  $(s, e, s') \in b^\bullet$ , and hence  $b \in pre(e)$ . Thus  $\gamma pre_c(e) \subseteq pre(e)$ . Suppose  $c_1, c_2 \in pre_c(e)$  with  $c_1\gamma b$  and  $c_2\gamma b$ . Considering an arbitrary transition  $(s, e, s')$ , we must have  $(s, *, s) \in c_1 \cap c_2$ . As  $c_1, c_2 \in \text{conn}(b)$ , by lemma 10.3.6,  $c_1 = c_2$ .

A similar argument holds for postconditions, and  $(\gamma, 1_E)$  fulfils the requirements of a relational morphism.

- (ii) The proof relies on a simple fact about relational morphisms, which is a direct consequence of proposition 10.3.10:

Let  $(\beta, \eta) : N \rightarrow N'$  be a morphism in **Safe**. If  $M$  is a reachable marking of  $N$ , then  $\beta M$  is a reachable marking of  $N'$  such that  $b_1\beta b'$  and  $b_2\beta b'$  implies  $b_1 = b_2$  for all conditions  $b_1, b_2 \in M$  and  $b'$  of  $N'$ .

We return to the proof of (ii). Let  $(\beta, \eta) : an_c(T) \rightarrow N$  be a morphism in **Safe**. For  $p$  a condition of  $N$ , we claim that

$$\{c \mid c\beta p\}$$

is a disjoint family of connected conditions. To establish the claim assume that  $c_1\beta p$  and  $c_2\beta p$ , where  $c_1 \cap c_2$  is nonempty and so necessarily contains  $(s, *, s)$ , for some state  $s$  of  $T$ . Then  $c_1, c_2 \in M_c(s)$ , where  $M_c(s)$  is a reachable marking of  $an_c(T)$ , by lemma 10.3.8(ii). Now, by the fact observed above, we conclude that  $c_1 = c_2$ , justifying the claim.

Thus we can define a relation  $\varphi$  between conditions of  $an_0(T)$  and  $N$  by taking  $\varphi^{op}$  as the partial function which, for  $p$  a condition of  $N$ , yields

$$\varphi^{op}(p) = \begin{cases} \bigcup \{c \mid c\beta p\} & \text{if nonempty,} \\ \text{undefined} & \text{otherwise} \end{cases}$$

—as remarked earlier, the union of a nonempty disjoint family of conditions is a condition. That  $(\varphi, \eta) : an_0(t) \rightarrow N$  is a morphism in **N** follows straightforwardly from  $(\beta, \eta) : an_c(T) \rightarrow N$  being a morphism in **Safe**. In order for the above diagram to commute we require that  $\beta = \varphi \circ \gamma$ , i.e.

$$\begin{aligned} c\beta p &\Leftrightarrow c(\varphi \circ \gamma)p \\ &\Leftrightarrow \exists b. c\gamma b \ \& \ b\varphi p \\ &\Leftrightarrow c \in \text{conn}(\varphi^{op}(p)) \text{ with } \varphi^{op}(p) \text{ defined.} \end{aligned}$$

But this shows that  $\varphi$  is determined uniquely. ■

**Corollary 10.3.14.** *The functors  $an_c : \mathbf{A}_c \rightarrow \mathbf{Safe}$  and  $na_c : \mathbf{Safe} \rightarrow \mathbf{A}_c$  form a coreflection with  $an_c$  left adjoint to  $na_c$ .*

**Proof.** The natural bijection required for the adjunction follows by composing the bijections of the two adjunctions from  $\mathbf{A}_c$  to  $\mathbf{A}_0$  with right adjoint  $V$ , and from  $\mathbf{A}_0$  to **N** with left and right adjoint  $an_0, na_0$  respectively, with the bijection of the previous lemma 10.3.13; letting  $T$  be an object in  $\mathbf{A}_c$  and  $N$  in **Safe**, there is a chain of bijections between morphisms:

$$T \rightarrow na_c(N) = V \circ na_0(N) \text{ in } \mathbf{A}_c,$$

$$T \rightarrow na_0(N) \text{ in } \mathbf{A}_0,$$

$$an_0(T) \rightarrow N \text{ in } \mathbf{N}, \text{ and}$$

$$an_c(T) \rightarrow N \text{ in } \mathbf{Safe}.$$

That the adjunction from  $\mathbf{A}_c$  to **Safe**, with left and right adjoints  $an_c, na_c$  respectively, is a coreflection follows from lemma 10.3.8. ■

So certain asynchronous transition systems are again in coreflection with a category of safe nets, but this time with a definition of morphism different from that in section 9.1. The neutral position of asynchronous transition systems with respect to which definition of morphism on nets is



taken, argues for their central role as models for concurrency. The coreflection from event structures to asynchronous transition systems cuts down to one  $\mathbf{E} \hookrightarrow \mathbf{A}_c$ . It composes with that to safe nets to yield a coreflection

$$\mathbf{E} \hookrightarrow \mathbf{A}_c \hookrightarrow \mathbf{Safe}.$$

The composite left adjoint is the construction of an “occurrence net” from an event structure given in [Nielsen *et al.*, 1981] (with the addition of a solitary marked condition)—see [Winskel, 1987a].

## 11 Semantics

In this section we show how to extend the models to include labels so that they can be used in giving semantics to process languages such as that of section 3. The denotational semantics involves a use of direct limits to handle recursively defined processes. The direct limits are with respect to embedding morphisms in the various categories. In many cases they can be replaced by a simpler treatment based on inclusion morphisms. We conclude by giving an operational semantics equivalent to a denotational semantics using labelled asynchronous transition systems. As will be seen, the operational semantics is obtained by expanding the rules of section 3.2, which generate the transitions, to include extra rules which express the independence between transitions.

### 11.1 Embeddings

The noninterleaving models, nets, asynchronous transition systems, trace languages, and event structures support recursive definitions. The idea of one process approximating another is caught in the notion of an embedding, a suitable kind of monomorphism with respect to which the categorical operations we have seen are continuous, in the sense of preserving  $\omega$ -colimits. This means that solutions of recursive definitions can be constructed as described for instance in [Barr and Wells, 1990]. Recall that the least fixed point  $\text{fix} F$  of a continuous functor  $F : \mathbf{X} \rightarrow \mathbf{X}$ , on a category  $\mathbf{X}$  with all  $\omega$ -colimits and initial object  $I$ , is constructed as the colimit of

$$I \xrightarrow{!} F(I) \xrightarrow{F(!)} F^2(I) \xrightarrow{F^2(!)} \dots \xrightarrow{F^{(n-1)}(!)} F^n(I) \xrightarrow{F^n(!)} \dots$$

where the morphism  $! : I \rightarrow F(I)$  is determined uniquely by the initiality of  $I$ .

In fact, for all models but nets, it suffices to restrict to inclusion-embeddings, embeddings based on inclusions, which form a large complete partial order. Fortunately the embeddings appropriate for different models are all related to each other. In the case of event structures the embeddings have already been introduced and studied by Kahn and

Plotkin under the name *rigid embeddings* (see [Kahn and Plotkin, 1979; Winskel, 1987a]).

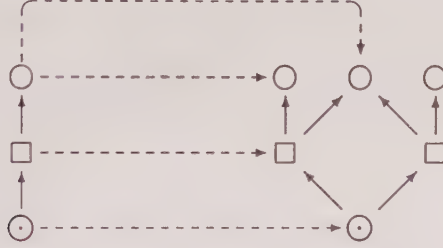
**Petri nets:** We first consider *embeddings* between nets. These are simply monomorphisms in the category  $\mathbf{N}$ .

**Definition 11.1.1.** An *embedding* of nets consists of a morphism of nets

$$(\beta, \eta) : N_0 \rightarrow N_1$$

such that  $\eta$  is an injective function and  $\beta^{op}$  is surjective, in the sense that for any condition  $b_0$  of  $N_0$  there is a condition  $b_1$  of  $N_1$  for which  $b_0 \beta b_1$ .

**Example 11.1.2.** Injection functions of a sum such as



are examples of embeddings between nets. The need to include such injections is a chief reason for allowing that part of a net morphism which relates conditions not to be injective. (Note too that there is no “projection morphism” sending  $e_1$  to  $e_0$  and  $e_2$  to undefined.)

Net embeddings are complete with respect to  $\omega$ -colimits. They have as an initial object the net consisting of a simple marked condition (which coincides with the initial object in the fuller category  $\mathbf{N}$ ). The existence of  $\omega$ -colimits is shown explicitly in the following construction:

**Proposition 11.1.3.** *Let*

$$N_0 \xrightarrow{(\beta_1, \eta_1)} N_1 \xrightarrow{(\beta_2, \eta_2)} \dots \xrightarrow{(\beta_k, \eta_k)} N_k \xrightarrow{(\beta_{k+1}, \eta_{k+1})} \dots \quad (\dagger)$$

be an  $\omega$ -chain of embeddings between nets  $N_k = (B_k, M_k, E_k, pre_k, post_k)$ , for  $k \in \omega$ .

Define  $N = (B, M, E, pre, post)$  where:

- $B$  consists of  $\omega$ -sequences

$$(b_0, b_1, \dots, b_k, \dots)$$

where  $b_k \in B_k \cup \{*\}$  such that  $b_k = \beta_{k+1}^{op}(b_{k+1})$  for all  $k \in \omega$ , with

the property that  $b_m \in B_m$  for some  $m \in \omega$ ; the initial marking  $M$  consists of all such sequences for which  $b_0 \in M_0$ .

- $E$  consists of  $\omega$ -sequences

$$(e_0, e_1, \dots, e_k, \dots)$$

where  $e_k \in E_k \cup \{*\}$  such that  $e_k \neq *$  implies  $\eta_{k+1}(e_k) = e_{k+1}$  for all  $k \in \omega$ , with the property that  $e_m \in E_m$  for some  $m \in \omega$ .

- The maps  $\text{pre} : E \rightarrow \text{Pow}(B)$  and  $\text{post} : E \rightarrow \text{Pow}(B)$  satisfy

$$\begin{aligned} b \in \text{pre}(e) &\Leftrightarrow \forall k \in \omega. (e_k \neq * \Rightarrow (b_k \neq * \ \& \ b_k \in \text{pre}_k(e_k))) \\ b \in \text{post}(e) &\Leftrightarrow \forall k \in \omega. (e_k \neq * \Rightarrow (b_k \neq * \ \& \ b_k \in \text{post}_k(e_k))), \end{aligned}$$

where we use  $e_k$  and  $b_k$  for the  $k$ -th components of the sequences  $e$  and  $b$  respectively.

Then  $N$  is a net. For each  $k \in \omega$ , the pair  $f_k = (\gamma_k, \epsilon_k)$  consisting of a relation  $\gamma_k \subseteq B \times B_k$  such that

$$c\gamma_k b \Leftrightarrow c = b_k$$

and a function  $\epsilon_k : E_k \rightarrow E$  such that

$$\epsilon_k(e') = e \Leftrightarrow e' = e_k$$

is an embedding of nets  $f_k : N_k \rightarrow N$ . Furthermore,  $N$  and the collection of embeddings  $f_k$ ,  $k \in \omega$ , is a colimit of the  $\omega$ -chain  $(\dagger)$ .

**Asynchronous transition systems:** An embedding between asynchronous transition systems consists of a monomorphism which reflects the independence relation.

**Definition 11.1.4.** An embedding of asynchronous transition systems consists of a morphism

$$(\sigma, \eta) : T_0 \rightarrow T_1,$$

between asynchronous transition systems  $T_0$  and  $T_1$  with independence relations  $I_0, I_1$  respectively, such that  $\sigma$  and  $\eta$  are injective and

$$\eta(e_0), \eta(e_1) \text{ defined } \& \ \eta(e_0)I_1\eta(e_1) \Rightarrow e_0 I_0 e_1$$

for any events  $e_0, e_1$  of  $T_0$ .

**Proposition 11.1.5.**

- If  $f : N_0 \rightarrow N_1$  is an embedding of nets, then  $na(f) : na(N_0) \rightarrow na(N_1)$  is an embedding of asynchronous transition systems. Moreover,  $na$  preserves  $\omega$ -colimits of embeddings.

- (ii) If  $g : T_0 \rightarrow T_1$  is an embedding of asynchronous transition systems, then  $an(g) : an(T_0) \rightarrow an(T_1)$  is an embedding of nets. Moreover,  $an$  preserves  $\omega$ -colimits of embeddings.

The operations on asynchronous transition systems we have seen are all continuous with respect to an order based on embeddings which are inclusions:

**Definition 11.1.6.** Let  $T_0 = (S_0, i_0, E_0, I_0, tran_0)$  and  $T_1 = (S_1, i_1, E_1, I_1, tran_1)$  be asynchronous transition systems. Define  $T_0 \trianglelefteq T_1$  iff  $S_0 \subseteq S_1$ ,  $E_0 \subseteq E_1$ , and  $(\sigma, \eta)$  is an embedding where  $\sigma$  is the inclusion  $S_0 \hookrightarrow S_1$  and  $\eta$  the inclusion  $E_0 \hookrightarrow E_1$ .

Asynchronous transition systems have  $\omega$ -colimits of embeddings. In particular, if

$$T_0 \trianglelefteq \dots \trianglelefteq T_n \trianglelefteq \dots$$

is an  $\omega$ -chain of asynchronous transition systems  $T_n = (S_n, i_n, E_n, I_n, tran_n)$ , it has a least upper bound

$$\left( \bigcup_{n \in \omega} S_n, i_0, \bigcup_{n \in \omega} E_n, \bigcup_{n \in \omega} I_n, \bigcup_{n \in \omega} tran_n \right)$$

which is not only an  $\omega$ -colimit in the category of inclusion-embeddings, but also in the category of embeddings. The situation restricts to asynchronous transition systems in  $\mathbf{A}_0$ ; they are closed under least upper bounds of  $\omega$ -chains under  $\trianglelefteq$ .

**Trace languages:** Embeddings on asynchronous transition systems induce embeddings on trace languages via the identification given by  $tla$ :

**Definition 11.1.7.** An *embedding* of trace languages consists of a morphism  $\eta : T \rightarrow T'$  of trace languages  $T, T'$ , with independence relations  $I, I'$  respectively, such that  $\eta$  is injective and

$$\eta(a), \eta(b) \text{ defined } \& \eta(a)I'\eta(b) \Rightarrow aIb, \quad \text{for all } a, b \in E.$$

Let  $T = (M, E, I), T' = (M', E', I')$  be trace languages. Define  $T \trianglelefteq T'$  iff

$$\begin{aligned} M &\subseteq M', \\ E &\subseteq E', \text{ and} \\ aIb &\Leftrightarrow aI'b, \quad \text{for all } a, b \in E. \end{aligned}$$

Again, embeddings and inclusion-embeddings have colimits of  $\omega$ -chains which in the case of inclusion-embeddings are given by unions. The functors  $atl$  and  $tla$  are continuous with respect to inclusion-embeddings.

**Event structures:** To treat recursively defined event structures we use a notion of embedding equivalent to that of the *rigid embeddings* of Kahn and Plotkin (see [Kahn and Plotkin, 1979; Winskel, 1987a]). Note that in the case of event structures (though not for the other models of this section) embeddings are always associated with projection morphisms in the opposite direction. When the embeddings are inclusions they amount to a substructure relation on event structures.

**Definition 11.1.8.** An *embedding* of event structures consists of a morphism  $\eta : ES_0 \rightarrow ES_1$  between event structures  $ES_0, ES_1$  where  $\eta$  is injective and such that its opposite, the partial function  $\eta^{op}$ , is a morphism of event structures  $\eta^{op} : ES_1 \rightarrow ES_0$ .

Let  $ES_0 = (E_0, \leq_0, \#_0)$ ,  $ES_1 = (E_1, \leq_1, \#_1)$  be event structures. Define  $ES_0 \sqsubseteq ES_1$  iff

$$E_0 \subseteq E_1,$$

$$\forall e \in E_1. e \leq_0 e_0 \Leftrightarrow e \leq_1 e_0,$$

for all  $e_0 \in E_0$ , and

$$e \#_0 e' \text{ iff } e \#_1 e',$$

for all  $e, e' \in E_0$ .

The  $\sqsubseteq$  order on event structures is a special case of the order on trace languages:

**Proposition 11.1.9.**

- (i) If  $ES \sqsubseteq ES'$ , for event structures  $ES, ES'$ , then  $etl(ES) \sqsubseteq etl(ES')$ , for the associated trace languages. Moreover,  $etl$  preserves  $\omega$ -colimits of inclusion-embeddings.
- (ii) If  $T \sqsubseteq T'$ , for trace languages  $T, T'$ , then  $tle(T) \sqsubseteq tle(T')$ , for the associated event structures. Moreover,  $tle$  preserves  $\omega$ -colimits of inclusion-embeddings.

## 11.2 Labelled structures

For noninterleaving models of concurrency like event structures, we distinguish between events, which carry the independence structure, and labels of the kind one sees in process algebras, whose use is to specify the nature of events, to determine for example how they synchronise. The denotation of a process, for example from the process language **Proc**, will most naturally be a labelled structure. The models we consider possess a set of events to which we can attach a labelling function. The set of events of an object  $X$  in a typical category  $\mathbf{X}$  of structures (for example,  $\mathbf{X}$  could be the category of event structures) is given by a functor  $E : \mathbf{X} \rightarrow \mathbf{Set}_*$ . This permits us to adjoin labelling sets to several different categories of models in the same way, using the following construction:



**Definition 11.2.1.** Let  $E : \mathbf{X} \rightarrow \mathbf{Set}_*$  be a functor from a category  $\mathbf{X}$ . Define  $\mathcal{L}(\mathbf{X})$  to be the category consisting of

*objects*  $(X, l : E(X) \rightarrow L)$  where  $X$  is an object of  $\mathbf{X}$  and  $l$  is a morphism in  $\mathbf{Set}$ , and

*morphisms* pairs  $(f, \lambda) : (X, l : E(X) \rightarrow L) \rightarrow (X', l' : E(X') \rightarrow L')$  where  $f : X \rightarrow X'$  in  $\mathbf{X}$  and  $\lambda : L \rightarrow L'$  in  $\mathbf{Set}$  satisfy

$$l' \circ E(f) = \lambda \circ l,$$

with composition defined coordinatewise, i.e.  $(f', \lambda') \circ (f, \lambda) = (f' \circ f, \lambda' \circ \lambda)$  provided  $f' \circ f$  and  $\lambda' \circ \lambda$  are defined.

To understand how this construction is used, take  $\mathbf{X}$  to be one kind of model, say event structures, so  $\mathbf{X}$  is  $\mathbf{E}$ . Then understanding  $E$  to be the forgetful functor to sets of events and partial functions has the effect of adjoining to event structures extra structure in the form of total labelling functions on events: the objects of the category  $\mathcal{L}(\mathbf{E})$  are labelled event structures  $(ES, l : E \rightarrow L)$  where  $ES$  is an event structure and  $l$  is a total function from its events  $E$  to a set of labels  $L$ ; morphisms  $(ES, l : E \rightarrow L) \rightarrow (ES', l' : E' \rightarrow L')$  are pairs  $(\eta, \lambda)$ , with  $\eta : ES \rightarrow_* ES'$  a morphism of event structures, and  $\lambda : L \rightarrow_* L'$  such that  $l' \circ \eta = \lambda \circ l$ .

Products and coproducts in  $\mathcal{L}(\mathbf{E})$  are obtained from the corresponding constructions in the unlabelled category because of the following general facts:

**Proposition 11.2.2.** Let  $E : \mathbf{X} \rightarrow \mathbf{Set}_*$  be a functor from a category  $\mathbf{X}$ . Assume  $\mathbf{X}$  has products. Then, a product of  $(X_0, l_0 : E(X_0) \rightarrow L_0)$  and  $(X_1, l_1 : E(X_1) \rightarrow L_1)$  in  $\mathcal{L}(\mathbf{X})$  is given by  $(X, l : E(X) \rightarrow L)$  with projections  $(\eta_0, \lambda_0)$ ,  $(\eta_1, \lambda_1)$ , where

- $X$  is a product of  $X_0, X_1$  in  $\mathbf{X}$  with projections  $\eta_0 : X \rightarrow X_0, \eta_1 : X \rightarrow X_1$
- $L$  is a product of  $L_0, L_1$  in  $\mathbf{Set}_*$  with projections  $\lambda_0 : L \rightarrow L_0, \lambda_1 : L \rightarrow L_1$
- $l = \langle l_0 \circ E(\eta_0), l_1 \circ E(\eta_1) \rangle : E(X) \rightarrow L$  is the unique mediating morphism to the product  $L$  such that  $\lambda_0 \circ l = l_0 \circ E(\eta_0)$  and  $\lambda_1 \circ l = l_1 \circ E(\eta_1)$ .

**Proposition 11.2.3.** Let  $E : \mathbf{X} \rightarrow \mathbf{Set}_*$  be a functor from a category  $\mathbf{X}$ . Assume  $\mathbf{X}$  has coproducts preserved by  $E$ . Then, a coproduct of  $(X_0, l_0 : E(X_0) \rightarrow L_0)$  and  $(X_1, l_1 : E(X_1) \rightarrow L_1)$  in  $\mathcal{L}(\mathbf{X})$  is given by  $(X, l : E(X) \rightarrow L)$  with injections  $(\eta_0, \lambda_0)$ ,  $(\eta_1, \lambda_1)$ , where

- $X$  is a coproduct of  $X_0, X_1$  in  $\mathbf{X}$  with injections  $\eta_0 : X_0 \rightarrow X, \eta_1 : X_1 \rightarrow X$

- $L$  is a coproduct of  $L_0, L_1$  in  $\mathbf{Set}_*$  with injections  $\lambda_0 : L_0 \rightarrow L, \lambda_1 : L_1 \rightarrow L$
- $l = [\lambda_0 \circ l_0, \lambda_1 \circ l_1] : E(X) \rightarrow L$  is the unique mediating morphism from the coproduct  $E(X)$  such that  $\lambda_0 \circ l_0 = l \circ E(\eta_0)$  and  $\lambda_1 \circ l_1 = l \circ E(\eta_1)$ .

There is a functor  $p : \mathcal{L}(\mathbf{X}) \rightarrow \mathbf{Set}_*$ ; a morphism of labelled structures

$$(f, \lambda) : (X, l : E(X) \rightarrow L) \rightarrow (X', l' : E(X') \rightarrow L')$$

is sent to

$$\lambda : L \rightarrow_* L'.$$

For any total function  $\lambda : L \rightarrow L'$  in  $\mathbf{Set}_*$ , this functor does have a strong cocartesian lifting of  $\lambda$  with respect to any object  $(X, l : E(X) \rightarrow L)$  in  $\mathcal{L}(\mathbf{X})$ : it is given by the morphism

$$(1_X, \lambda) : (X, l : E(X) \rightarrow L) \rightarrow (X, \lambda \circ l : E(X) \rightarrow L')$$

in  $\mathcal{L}(\mathbf{X})$ . This yields a relabelling operation when  $\mathbf{X}$  is specialised to one of the models.

For any of the models, there are also strong cartesian liftings of inclusions  $L \hookrightarrow L'$  in  $\mathbf{Set}_*$  with respect to a labelled structure  $(X, l : E(X) \rightarrow L)$ , though this requires an argument resting on the fact that the categories of structures (without labels) that we consider support an operation of restriction to a prescribed subset of events. For example, given an event structure  $ES = (E', \leq', \#')$  and a specified subset  $E \subseteq E'$ , there is an event structure, the *restriction* of  $ES$  to  $E$  gives an event structure  $(E_0, \leq, \#)$  as follows:

its set of events consists of  $E_0 = \{e \in E \mid \forall e' \leq e. e' \in E\}$ ;

its causal dependency relation satisfies

$$e \leq e' \Leftrightarrow e, e' \in E_0 \text{ \& } e \leq' e';$$

its conflict relation satisfies

$$e \# e' \Leftrightarrow e, e' \in E_0 \text{ \& } e \# e'.$$

To treat recursion on labelled structures we extend embeddings to labelled structures, such as  $\mathcal{L}(\mathbf{E})$ . A morphism of labelled structures

$$(f, \lambda) : (X, l : E \rightarrow L) \rightarrow (X', l' : E' \rightarrow L')$$

is taken to be an embedding (or an inclusion-embedding) if  $f : X \rightarrow X'$  is an embedding (or an inclusion-embedding) and  $\lambda$  is an inclusion of sets.

The labelled structures have colimits of  $\omega$ -chains formed from colimits of the unlabelled structures. In particular, a chain

$$(X_0, l_0) \leq \cdots \leq (X_n, l_n) \leq \cdots$$

of labelled structures  $(X_n, l_n : E_n \rightarrow L_n)$  has least upper bound  $(\bigcup_{n \in \omega} X_n, \bigcup_{n \in \omega} l_n)$ . The labelling function  $\bigcup_{n \in \omega} l_n$  has domain  $\bigcup_{n \in \omega} E_n$  and co-domain  $\bigcup_{n \in \omega} L_n$ .

In section 3 we had to go to a little trouble to extend the restriction and relabelling operations to all transition systems regardless of their labelling set. In general, some care is needed in making functors with respect to embeddings out of some of the operations. The operations of restriction and relabelling  $(- \upharpoonright \Lambda)$  and  $(-\{\Xi\})$  yield functors on categories of embeddings. Suppose there is an embedding  $f : X \rightarrow X'$  between labelled structures  $X, X'$  with labelling sets  $L, L'$  respectively, necessarily related by an inclusion  $L \hookrightarrow L'$ . The structure  $X$  with labelling set  $L$  restricts to  $X \upharpoonright \Lambda$  associated with a particular cartesian lifting

$$c : X \upharpoonright \Lambda \rightarrow X$$

of the inclusion  $L \cap \Lambda \hookrightarrow L$ . Similarly,  $X'$  is associated with the cartesian lifting

$$c' : X' \upharpoonright \Lambda \rightarrow X'$$

of the inclusion  $L' \cap \Lambda \hookrightarrow L'$ . Because  $c'$  is strong cartesian there is a unique morphism

$$(f \upharpoonright \Lambda) : X \upharpoonright \Lambda \rightarrow X' \upharpoonright \Lambda$$

projecting to the inclusion  $L \cap \Lambda \hookrightarrow L' \cap \Lambda$  such that  $f \circ c = c' \circ (f \upharpoonright \Lambda)$ . This ensures that  $(- \upharpoonright \Lambda)$  is a functor from the subcategory with embeddings. Moreover, for each model,  $(f \upharpoonright \Lambda)$  is an (inclusion-)embedding provided  $f$  is. In a similar way a relabelling function  $\Xi$  is associated with cocartesian liftings  $X \rightarrow X\{\Xi\}$  of  $L \rightarrow \Xi L$  for any structure  $X$  with labelling set  $L$ , and gives rise to a functor with respect to embeddings. For all the models here, it is a straightforward matter to define a prefixing operation on the various labelled structures so that it is continuous with respect to a given choice of embedding. The labelled versions of continuous functors are continuous.

The various categories of labelled structures, such as  $\mathcal{L}(\mathbf{E})$  for example, provide a semantics to the process language **Proc** interpreting constructions in the process language as the appropriate universal construction, so abstractly this proceeds exactly as in section 3.

## 11.3 Operational semantics

### 11.3.1 Transition systems with independence

The model of asynchronous transition systems is based on events which carry an independence relation. The nature of these events can then be

specified by a further level of labelling. There is an alternative, more direct, presentation of (certain kinds of) labelled asynchronous transition systems, got by extending transition systems with an independence relation on its transitions. Transition systems with independence are definable by the techniques of structural operational semantics in a way which directly extends that of section 3.

**Definition 11.3.1.** A transition system with independence<sup>10</sup> is defined to be a structure

$$(S, i, L, Tran, I)$$

where  $(S, i, L, Tran)$  is a transition system and the independence relation  $I \subseteq Tran^2$  is an irreflexive, symmetric relation, such that

- (1)  $(s, a, s_1) \sim (s, a, s_2) \Rightarrow s_1 = s_2$
- (2)  $(s, a, s_1)I(s, b, s_2) \Rightarrow \exists u. (s, a, s_1)I(s_1, b, u) \ \& \ (s, b, s_2)I(s_2, a, u)$
- (3)  $(s, a, s_1)I(s_1, b, u) \Rightarrow \exists s_2. (s, a, s_1)I(s, b, s_2) \ \& \ (s, b, s_2)I(s_2, a, u)$
- (4)
  - (i)  $(s, a, s_1) \prec (s_2, a, u)I(w, b, w') \Rightarrow (s, a, s_1)I(w, b, w')$
  - (ii)  $(w, b, w')I(s, a, s_1) \prec (s_2, a, u) \Rightarrow (w, b, w')I(s_2, a, u)$

where the relation  $\prec$  between transitions is defined by

$$(s, a, s_1) \prec (s_2, a, u) \Leftrightarrow \exists b. (s, a, s_1)I(s, b, s_2) \ \& \ (s, a, s_1)I(s_1, b, u) \ \& \ (s, b, s_2)I(s_2, a, u),$$

and  $\sim$  is the least equivalence relation including  $\prec$ .

As morphisms on transition systems with independence we take morphisms on the underlying transition systems which preserve independence, *i.e.* a morphism  $(\sigma, \lambda) : T \rightarrow T'$  should satisfy

If  $(s, a, s')$  and  $(u, b, u')$  are independent transitions of  $T$  and  $\lambda(a)$  and  $\lambda(b)$  are both defined, then  $(\sigma(s), \lambda(a), \sigma(s'))$  and  $(\sigma(u), \lambda(b), \sigma(u'))$  are independent transitions of  $T'$ .

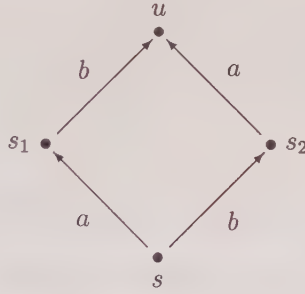
Composition is inherited from that in **T**. We write **TI** for the category of transition systems with independence.

Thus transition systems with independence are precisely what their name suggests, *viz.* transition systems of the kind used to model languages

---

<sup>10</sup>Axiom 2 of transition systems with independence is not essential to much of the development. It ensures that the trace language of a transition system with independence is coherent, so that the associated event structure has the property that conflict is determined in a binary fashion.

like CCS and CSP but with an additional relation expressing when one transition is independent of another. The axioms (2) and (3) in the definition above describe intuitive properties of independence, similar to those we have seen. The relation  $\prec$  expresses when two transitions represent occurrences of the same event. This relation extends to an equivalence relation  $\sim$  between transitions; the equivalence classes  $\{(s, a, s')\}_\sim$ , of transitions  $(s, a, s')$ , are the events of the transition system with independence. Property (4) is then seen as asserting that the independence relation respects events. Note that property (4) implies that if  $(s, a, s_1) \prec (s_2, a, u)$ , *i.e.* there is a “square” of transitions



with

$$(s, a, s_1)I(s, b, s_2) \ \& \ (s, a, s_1)I(s_1, b, u) \ \& \ (s, b, s_2)I(s_2, a, u),$$

then we also have the independence

$$(s_1, b, u)I(s_2, a, u).$$

The first property (1) simply says that the occurrence of an event at a state yields a unique state. Note that property (1) implies the uniqueness of the states,  $u$  and  $s_2$ , whose existence is asserted by properties (2) and (3) respectively.

In this way a transition system with independence can be viewed as an asynchronous transition system in which the events are labelled, an event  $\{(s, a, s')\}_\sim$  carrying the label  $a$ . The resulting asynchronous transition system is *extensional* in that it has the property that

- (1) for any label there is at most one event with that label involved in a transition between two states.

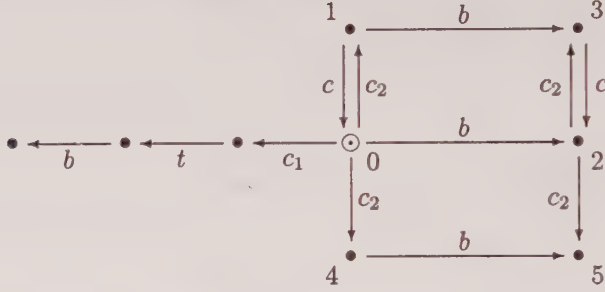
It is special in another way too. An asynchronous transition system can be regarded as a transition system with independence, in which the independence on transitions is induced by that on its events. The asynchronous transition systems which result from transition systems with independence have the special property that



(2) the map  $\{(s, e, s')\}_\sim \mapsto e$  is a bijection.

There is in fact an equivalence between the category of transition systems with independence and the full subcategory of  $\mathcal{L}(\mathbf{A})$  for which the objects are labelled transition systems with the properties (1) and (2).

Let's return to our example of 3.3. It is now an easy matter to extend the transition-system semantics there to take account of independence. We simply specify which transitions are independent of which others. Copying the transition system of 3.3,



we assert in addition the following independencies

$$(0, c_2, 1)I(0, b, 2), \quad (1, c, 0)I(1, b, 3), \quad (0, b, 2)I(0, c_2, 4)$$

which then generate others by the axioms in the definition of a transition system with independence.

### 11.3.2 Operational rules

Transition systems with independence have the feature that they are definable by structural operational semantics in much the same way as transition systems, but with the usual rules for transitions being supplemented by rules specifying the independence relation between transitions.

To motivate the rules we first examine how the product lends itself readily to a presentation via rules of structural operational semantics. Assume  $T_0 = (S_0, i_0, L_0, Tran_0, I_0)$  and  $T_1 = (S_1, i_1, L_1, Tran_1, I_1)$  are transition systems with independence. Their categorical product  $T_0 \times T_1$  is  $(S, i, L, Tran, I)$  where  $(S, i, L, Tran)$  is the product of the underlying transition systems  $(S_0, i_0, L_0, Tran_0)$ ,  $(S_1, i_1, L_1, Tran_1)$ , with projections  $(\rho_0, \pi_0)$ ,  $(\rho_1, \pi_1)$ , and the independence relation  $I$  on transitions is given by

$$(s, a, s')I(u, b, u') \text{ iff}$$

$$\pi_0(a), \pi_0(b) \text{ defined} \Rightarrow (\rho_0(s), \pi_0(a), \rho_0(s'))I_0(\rho_0(u), \pi_0(b), \rho_0(u')) \ \&$$

$$\pi_1(a), \pi_1(b) \text{ defined} \Rightarrow (\rho_1(s), \pi_1(a), \rho_1(s'))I_1(\rho_1(u), \pi_1(b), \rho_1(u')).$$

The characterisation of the independence relation can be simplified through the use of idle transitions. An independence relation like  $I \subseteq Tran \times Tran$

extends to a relation  $I_* \subseteq Tran_* \times Tran_*$  in which

$$(s, a, s')I_*(u, b, u') \Leftrightarrow a = * \text{ or } b = * \text{ or } (s, a, s')I(u, b, u').$$

An idle transition is thus always independent of any transition, idle or otherwise. Now we have the simplification:

$$\begin{aligned} (s, a, s')I_*(u, b, u') \text{ iff} \\ (\rho_0(s), \pi_0(a), \rho_0(s'))I_{0*}(\rho_0(u), \pi_0(b), \rho_0(u')) \ \& \\ (\rho_1(s), \pi_1(a), \rho_1(s'))I_{1*}(\rho_1(u), \pi_1(b), \rho_1(u')). \end{aligned}$$

We have already seen rules to give the transitions of the product (section 3.2). To define the product of transition systems with independence we adjoin the following rule, which reformulates the condition for two transitions of a product to be independent:

$$\frac{(s_0, a_0, s'_0)I_{0*}(u_0, b_0, u'_0), \quad (s_1, a_1, s'_1)I_{1*}(u_1, b_1, u'_1)}{((s_0, s_1), a_0 \times a_1, (s'_0, s'_1))I_*((u_0, u_1), b_0 \times b_1, (u'_0, u'_1))}.$$

Similarly, the fibre coproduct of transition systems with independence is given by the fibre coproduct of the underlying transition systems together with an independence relation inherited directly from the components. This too can be expressed by simple rules, which are essentially unchanged in the nondeterministic sum  $\oplus$ , where we first enlarge the labelling sets to their union and then form the fibre coproduct. Let  $T_0$  and  $T_1$  be the transition systems with independence above. Their sum  $T_0 \oplus T_1$  consists of a transition system, formed as the nondeterministic sum of their underlying transition systems associated with injection functions  $in_0, in_1$  on states, together with an independence relation satisfying

$$(s, a, s')I(u, b, u')$$

iff

$$\begin{aligned} [\exists s_0, s'_0, u_0, u'_0. \\ s = in_0(s_0) \ \& \ s' = in_0(s'_0) \ \& \ u = in_0(u_0) \ \& \ u' = in_0(u'_0) \\ \ \& \ (s_0, a, s'_0)I_0(u_0, b, u'_0)] \end{aligned}$$

or

$$\begin{aligned} [\exists s_1, s'_1, u_1, u'_1. \\ s = in_1(s_1) \ \& \ s' = in_1(s'_1) \ \& \ u = in_1(u_1) \ \& \ u' = in_1(u'_1) \\ \ \& \ (s_1, a, s'_1)I_1(u_1, b, u'_1)]. \end{aligned}$$

Expressed by rules the condition on the independence relation becomes

$$\frac{(s_0, a, s'_0)I_0(u_0, b, u'_0)}{(in_0(s_0), a, in_0(s'_0))I(in_0(u_0), b, in_0(u'_0))}$$

$$\frac{(s_1, a, s'_1)I_1(u_1, b, u'_1)}{(in_1(s_1), a, in_1(s'_1))I(in_1(u_1), b, in_1(u'_1))}.$$

As usual a restriction can be understood as a cartesian lifting of an inclusion morphism on labelling sets; there is an obvious functor from **TI** to **Set**<sub>\*</sub> projecting to the labelling sets and the labelling functions between them. Letting  $T = (S, i, L, Tran, I)$  be a transition system with independence, the restriction to a subset of labels  $\Lambda$  is

$$T \upharpoonright \Lambda = (S, i, L', Tran', I')$$

where  $L' = L \cap \Lambda$ ,  $Tran' = Tran \cap (S \times L' \times S)$ , and  $I' = I \cap (Tran' \times Tran')$ . Although this operation may change the  $\sim$  relation, increasing the number of events, it preserves the axioms required of a transition system with independence. The rule in the operational semantics for the independence relation of a restriction expresses that it is got simply by cutting down the original independence relation.

Relabelling is associated with a cocartesian lifting of the relabelling function on labelling sets. In defining it we can take advantage of a *unicity property* of those transition systems arising from the operational semantics:

Suppose  $s \xrightarrow{a} s'$  and  $s \xrightarrow{b} s'$  are transitions obtained from the operational semantics (version 2) of section 3.2. Then  $a = b$ .

This property is easily observed to be preserved by the rules.

Let  $T = (S, i, L, Tran, I)$  be a transition system with independence, assumed to satisfy the unicity property

$$(s, a, s') \in Tran \ \& \ (s, b, s') \in Tran \Rightarrow a = b.$$

For  $\Xi : L \rightarrow L'$  the relabelling

$$T\{\Xi\} = (S, i, L', Tran', I'),$$

where  $Tran' = \{(s, b, s') \mid \exists a. b = \Xi(a) \ \& \ (s, a, s') \in Tran\}$  and

$$(s, a, s')I'(t, b, t') \Leftrightarrow \exists a', b'. a = \Xi(a') \ \& \ b = \Xi(b') \ \& \ (s, a', s')I(t, b', t').$$

Because the transition system  $T$  satisfies the unicity property the construction  $T\{\Xi\}$  yields a transition system with independence (without the assumption of unicity the new relation  $I'$ , as defined, need not respect events, and a more complicated definition is needed). Consequently in the operational semantics we can get away with a rule which says that the independence relation of the relabelled transition system is simply the image of the original.

We obtain an operational semantics for **Proc** as a transition system with independence by extending version 2 of the rules of section 3 for the transitions between (tagged) states by the following rules for the independence relation (also relating idle transitions):

*Rules for independence*

$$(s, a, s') \text{ I } (u, *, u) \quad \frac{(s, a, s') \text{ I } (u, b, u')}{(u, b, u') \text{ I } (s, a, s')}$$

$$\frac{(s, a, s') \text{ I } (t, b, t')}{((n, s), a, (n, s')) \text{ I } ((n, t), b, (n, t'))}$$

*Sum:*

$$\frac{(s, a, s') \text{ I } (s, b, s'')}{(s \oplus t, a, (0, s')) \text{ I } (s \oplus t, b, (0, s''))} \quad a, b \neq *$$

$$\frac{(t, a, t') \text{ I } (t, b, t'')}{(s \oplus t, a, (1, t')) \text{ I } (s \oplus t, b, (1, t''))} \quad a, b \neq *$$

$$\frac{(s, a, s') \text{ I } (u, b, u')}{(s \oplus t, a, (0, s')) \text{ I } ((0, u), b, (0, u'))} \quad u \neq s, a \neq *$$

$$\frac{(t, a, t') \text{ I } (u, b, u')}{(s \oplus t, a, (1, t')) \text{ I } ((1, u), b, (1, u'))} \quad u \neq t, a \neq *$$

*Product:*

$$\frac{(s_1, a_1, s'_1) \text{ I } (s_2, a_2, s'_2) \quad (t_1, b_1, t'_1) \text{ I } (t_2, b_2, t'_2)}{(s_1 \times t_1, a_1 \times b_1, s'_1 \times t'_1) \text{ I } (s_2 \times t_2, a_2 \times b_2, s'_2 \times t'_2)}$$

*Restriction and relabelling:*

$$\frac{(s, a, s') \text{ I } (t, b, t')}{(s \upharpoonright \Lambda, a, s' \upharpoonright \Lambda) \text{ I } (t \upharpoonright \Lambda, b, t' \upharpoonright \Lambda)} \quad a, b \in \Lambda$$

$$\frac{(s, a, s') \text{ I } (t, b, t')}{(s\{\Xi\}, \Xi(a), s'\{\Xi\}) \text{ I } (t\{\Xi\}, \Xi(b), t'\{\Xi\})}$$

*Recursion:*

$$\frac{(t[\text{rec } x.t/x], a, s) \text{ } I \text{ } (t[\text{rec } x.t/x], b, u)}{(\text{rec } x.t, a, (2, s)) \text{ } I \text{ } (\text{rec } x.t, b, (2, u))} a, b \neq *$$

$$\frac{(t[\text{rec } x.t/x], a, s) \text{ } I \text{ } (u, b, u')}{(\text{rec } x.t, a, (2, s)) \text{ } I \text{ } ((2, u), b, (2, u'))} u \not\equiv t[\text{rec } x.t/x], a \neq *$$

A closed term of **Proc** determines a transition system with independence consisting of all those states and transitions forwards-reachable from it together with an independence relation determined by the rules above. Notice that there are no extra rules for prefixing because the transition immediately possible for a prefixed process is not independent of any other. The rules for product, restriction, and relabelling are straightforward reformulations as rules of the requirements on their independence relations. The rules for sum and recursion require further explanation. For a sum  $s \oplus t$ , taking the injection functions  $in_0, in_1$  on states to satisfy, for example,

$$in_0(s) = s \oplus t, \quad \text{and} \quad in_0(u) = (0, u) \text{ if } u \not\equiv s,$$

we can understand the rules for sum, together with the rule for tagged terms, as saying that independence for a sum is precisely that inherited separately from the components. Because the transition system is acyclic (lemma 3.2.2), there is an isomorphism between the transition system reachable from  $\text{rec } x.t$  and its unfolding  $t[\text{rec } x.t/x]$  (this fact is used earlier in the proof of theorem 3.2.3). The isomorphism is given by

$$\begin{array}{ll} \text{rec } x.t & \mapsto t[\text{rec } x.t/x] \\ (2, u) & \mapsto u. \end{array}$$

The rules for recursively defined processes, with the final rule for tagged terms, ensure that transitions reachable from  $\text{rec } x.t$  are independent precisely when their images under this isomorphism are independent.

A denotational semantics where denotations are transition systems with independence can be presented along standard lines; the categorical constructions defined above are used to interpret the operations.

We have already discussed the categorical constructions in **TI** which are used to interpret the operations of the process language. It remains to handle recursion. We define an appropriate ordering, with respect to which all the constructions are continuous:

**Definition 11.3.2.** Let  $T = (S, i, L, \text{tran}, I)$  and  $T' = (S', i', L', \text{tran}', I')$  be transition systems with independence. Define  $T \leq T'$  iff



$$S \subseteq S' \text{ with } i = i, L \subseteq L', \text{tran} \subseteq \text{tran}', \text{ and } \\ \forall (s, a, s'), (t, b, t') \in \text{tran}. (s, a, s')I(t, b, t') \Leftrightarrow (s, a, s')I'(t, b, t').$$

Now, as earlier in section 3, for straightforward transition systems, we can give denotations to recursively defined processes. The result is that with respect to an environment  $\rho$  assigning meanings to process variables as transition systems with independence, we can give the denotation of a process term  $t$  as a transition system with independence:

$$\mathbf{TI}[t]\rho.$$

The denotational semantics agrees with the operational semantics. The proof proceeds analogously to that of theorem 3.2.3—further details are given in Appendix C(b).

**Definition 11.3.3.** For  $T = (S, i, L, \text{Tran}, I)$  consisting of a transition system  $(S, i, L, \text{Tran})$  and relation  $I \subseteq \text{Tran} \times \text{Tran}$ , define  $\mathcal{R}(T)$  to be  $(S', i, L', \text{Tran}', I')$  consisting of states  $S'$  reachable from  $i$ , with initial state  $i$ , and transitions  $\text{Tran}' = \text{Tran} \cap (S' \times L \times S')$  with labelling set  $L'$  consisting of those labels appearing in  $\text{Tran}'$  and  $I' = I \cap (\text{Tran}' \times \text{Tran}')$ .

Assume  $t$  is a closed term of **Proc**. Let  $T$  consist of the transition system got from version 2 in section 3, with initial state  $t$ , with the independence relation given by the rules above. Define

$$\mathcal{Op}(t) = \mathcal{R}(T).$$

**Theorem 11.3.4.** Let  $t$  be a closed process term. Then, for any arbitrary environment  $\rho$ ,

$$\mathcal{Op}(t) \cong \mathcal{R}(\mathbf{TI}[t]\rho),$$

a label-preserving isomorphism.

The denotational semantics in **TI** is closely related to that in  $\mathcal{L}(\mathbf{A})$  which we write as  $\mathbf{A}[t]\rho$ , for a term  $t$  and an environment  $\rho$  interpreting variables in  $\mathcal{L}(\mathbf{A})$ . There is an obvious functor from  $\mathcal{L}(\mathbf{A})$  to **TI** (it is not, however, adjoint to that functor identifying a transition system with independence with an equivalent labelled asynchronous transition system). On objects it acts as follows:

**Definition 11.3.5.** Let  $T = (S, i, E, I, \text{Tran}, l : E \rightarrow L)$  be an object of  $\mathcal{L}(\mathbf{A})$ . Define  $u(T)$  to be  $(S, i, L, \text{Tran}', I')$  where

$$(s, a, s') \in \text{Tran}' \Leftrightarrow \exists e. l(e) = a \ \& \ (s, e, s') \in \text{Tran} \\ (s, a, s')I'(t, b, t') \Leftrightarrow \exists e_0, e_1. l(e_0) = a \ \& \ l(e_1) = b \ \& \ (s, e_0, s')I(t, e_1, t').$$

**Theorem 11.3.6.** Let  $t$  be a term of the process language **Proc**. For any environment  $\rho$  interpreting process variables in  $\mathcal{L}(\mathbf{A})$ ,

$$\mathbf{TI}[t](u \circ \rho) = u(\mathbf{A}[t]\rho).$$

**Proof.** The operation  $u$  can be shown to be continuous with respect to the orderings  $\leq$  and to preserve the operations of **Proc**. A structural induction on terms  $t$  of **Proc** shows that

$$\mathbf{TI}[t](u \circ \rho) = u(\mathbf{A}[t]\rho),$$

for an environment  $\rho$  interpreting variables in  $\mathcal{L}(\mathbf{A})$ . The case where  $t$  is a recursive process relies on the fact that if  $F$  and  $G$  are continuous functions on (large) cpo's **TI** and  $\mathcal{L}(\mathbf{A})$  respectively, ordered by  $\leq$ , such that

$$F \circ u = u \circ G$$

then, because  $u$  is continuous and preserves the bottom element in the definitions by recursion,

$$\text{fix} F = u(\text{fix} G).$$

■

As we will see, the coreflections between categories of unlabelled structures extend to categories of labelled structures. In particular, this yields a coreflection  $\mathcal{L}(\mathbf{E}) \hookrightarrow \mathcal{L}(\mathbf{A})$ . This coreflection cuts down to one  $\mathcal{L}(\mathbf{E}) \hookrightarrow \mathbf{TI}$  and semantics in  $\mathcal{L}(\mathbf{A})$  and **TI** unfold to the same semantics in labelled event structures.<sup>11</sup>

## 12 Relating models

Earlier in section 11.2, it was seen how to attach labels to events of structures in a uniform way. In relating semantics in terms of the different models, we shall also wish to extend functors between categories of models to functors between their labelled versions. For this we use the fact that the functors of interest are accompanied by natural transformations, so that a general scheme described in the following definition applies. The components of the natural transformation relate the event sets before and after application of the functor; for example, the natural transformation

---

<sup>11</sup>It follows that the labelled event structure obtained from the Petri net semantics is isomorphic to that got by “unfolding” the operational semantics. We can also ask about the following method for obtaining a labelled-net semantics directly from the semantics in **TI**. Certainly we can regard a transition system with independence as a labelled asynchronous transition system (how, is explained early in this section) and thus we can obtain a net via the adjunction between asynchronous transition systems and nets. At the time of writing, it is not decided whether or not this yields an asynchronous transition system in  $\mathbf{A}_0$ , and thus, via the coreflection, a net with the same underlying asynchronous transition system as its behaviour.

accompanying the functor from trace languages to event structures has components mapping the events of a trace language to their associated symbols in the alphabet.

**Definition 12.0.1.** Let  $E_C : \mathbf{C} \rightarrow \mathbf{Set}_*$  and  $E_D : \mathbf{D} \rightarrow \mathbf{Set}_*$  be functors (taking structures to their underlying event sets).

Suppose  $F : \mathbf{C} \rightarrow \mathbf{D}$  is a functor and  $\phi : E_D \circ F \rightarrow E_C$  is a natural transformation with components in  $\mathbf{Set}$  (the natural transformation relates the event sets resulting from the application of  $F$  to those originally). Define the functor  $\mathcal{L}(F, \phi) : \mathcal{L}(\mathbf{C}) \rightarrow \mathcal{L}(\mathbf{D})$  to act on objects so

$$(C, l : E_C(C) \rightarrow L) \mapsto (F(C), l \circ \phi_C : E_D \circ F(C) \rightarrow L)$$

and on morphisms so

$$(f, \lambda) \mapsto (F(f), \lambda)$$

where  $(f, \lambda) : (C, l : E_C(C) \rightarrow L) \rightarrow (C', l' : E_C(C') \rightarrow L')$ .

Under reasonable conditions the labelling operation  $\mathcal{L}(-)$  preserves adjunctions, coreflections, and reflections:

**Lemma 12.0.2.** Let  $E_C : \mathbf{C} \rightarrow \mathbf{Set}_*$  and  $E_D : \mathbf{D} \rightarrow \mathbf{Set}_*$  be functors.

Suppose  $F : \mathbf{C} \rightarrow \mathbf{D}$  is a functor and  $\phi : E_D \circ F \rightarrow E_C$  is a natural transformation. Suppose  $G : \mathbf{D} \rightarrow \mathbf{C}$  is a functor and  $\gamma : E_C \circ G \rightarrow E_D$  is a natural transformation. Suppose there is an adjunction with  $F$  left adjoint to  $G$ , with unit  $\eta$  and counit  $\epsilon$ .

If, for any  $C \in \mathbf{C}, D \in \mathbf{D}$ ,

$$1_{E_C(C)} = \phi_C \circ \gamma_{F(C)} \circ E_C(\eta_C) \quad \text{and} \quad E_D(\epsilon_D) = \gamma_D \circ \phi_{G(D)}, \quad (1)$$

then the functors  $\mathcal{L}(F, \phi) : \mathcal{L}(\mathbf{C}) \rightarrow \mathcal{L}(\mathbf{D})$  and  $\mathcal{L}(G, \gamma) : \mathcal{L}(\mathbf{D}) \rightarrow \mathcal{L}(\mathbf{C})$  form a fibrewise adjunction with  $\mathcal{L}(F, \phi)$  left adjoint to  $\mathcal{L}(G, \gamma)$  and unit and counit given as follows: the unit at  $(C, l : E_C(C) \rightarrow L)$  is  $(\eta_C, 1_L)$ ; the counit at  $(D, l : E_D(D) \rightarrow L)$  is  $(\epsilon_D, 1_L)$ . If, in addition, the adjunction between  $F$  and  $G$  is a coreflection or reflection, then  $\mathcal{L}(F, \phi)$  and  $\mathcal{L}(G, \gamma)$  form a coreflection or reflection respectively.

**Proof.** By [MacLane, 1971, theorem 2, p.81] the adjunction between  $\mathbf{C}$  and  $\mathbf{D}$ , is determined by the functors  $F, G$ , the natural transformations  $\eta, \epsilon$ , and the fact that the compositions

$$\begin{aligned} G(D) &\xrightarrow{\eta_G(D)} GFG(D) \xrightarrow{G(\epsilon_D)} G(D), \\ F(C) &\xrightarrow{F(\eta_C)} FGF(C) \xrightarrow{\epsilon_{F(C)}} F(C) \end{aligned}$$

are identities. The condition (1) is sufficient to ensure that these facts lift straightforwardly to the labelled categories and functors, determining an adjunction with unit and counit as claimed. The unit and counit are

vertical, making the adjunction fibrewise. Given their form, they become natural isomorphisms if  $\eta$  or  $\epsilon$  are; the property of being a coreflection or reflection is preserved by the construction. ■

This lemma enables us to transport the adjunctions that exist between categories of unlabelled structures to adjunctions between the corresponding categories of labelled structures. The role of the natural transformations is to relate the event sets of the image of a functor to the event set of the original object. We are only required to check that the natural transformations, tracking the functors in the labelling category of sets, relate well to the unit and counit, in the sense of (1) above.

As an example we consider how to extend the coreflection between event structures and trace languages to labelled versions of these structures using lemma 12.0.2. The role of the natural transformations in the lemma is to relate the event sets of the image of a functor to the event set of the original object, as can be seen by considering the functor

$$tle : \mathbf{TL} \rightarrow \mathbf{E}.$$

Let  $E_{TL} : \mathbf{TL} \rightarrow \mathbf{Set}_*$  be the forgetful functor from trace languages to their alphabets. Let  $E_E : \mathbf{E} \rightarrow \mathbf{Set}_*$  be the forgetful functor from event structures to their sets of events. A component of the counit of the coreflection between  $\mathbf{E}$  and  $\mathbf{TL}$  maps the events of a trace language to its alphabet. It yields a natural transformation  $\gamma : E_E \circ tle \rightarrow E_{TL}$ . A trace language  $T = (M, A, I)$  with labelling  $l : A \rightarrow L$  can now be sent to the event structure  $tle(T)$  with labelling  $l \circ \gamma_T : E \rightarrow L$ . This extends to a functor  $\mathcal{L}(tle, \gamma) : \mathcal{L}(\mathbf{TL}) \rightarrow \mathcal{L}(\mathbf{E})$ . The functor  $etl : \mathbf{E} \rightarrow \mathbf{TL}$  does not change the set of events and we associate it with the identity natural transformation  $1 : E_{TL} \circ etl \rightarrow E_E$ . These choices of natural transformations to associate with the functors  $etl$  and  $tle$  ensure that condition (1) of lemma 12.0.2 holds. To see this, we use the fact that

$$\epsilon_{etl(E)} \circ etl(\eta_E) = 1_{etl(E)}$$

obtains for counit  $\epsilon$  and unit  $\eta$  of the adjunction, for any  $E \in \mathbf{E}$ . Thus applying the functor  $E_{TL}$ , we get

$$E_{TL}(\epsilon_{etl(E)}) \circ E_{TL}(etl(\eta_E)) = E_{TL}(1_{etl(E)}).$$

But now, recalling how  $E_{TL}$  and  $etl$  act, we see that

$$\epsilon_{etl(E)} \circ \eta_E = 1_E,$$

i.e. the first half of (1) holds. The remaining half of (1) reduces to an obvious equality. We conclude by lemma 12.0.2 that



$$\mathcal{L}(etl, 1) : \mathcal{L}(\mathbf{E}) \rightarrow \mathcal{L}(\mathbf{TL})$$

forms a coreflection, with right adjoint  $\mathcal{L}(tle, \gamma)$ . The coreflection  $\mathbf{E} \hookrightarrow \mathbf{TL}$  cuts down to one  $\mathbf{E} \hookrightarrow \mathbf{TL}_0$ , which extends to labelled structures.

So, in particular, we can lift the coreflection between event structures and trace languages to labelled versions of these structures. In a similar, but much easier manner, we can lift the adjunction between  $\mathbf{A}$  and  $\mathbf{N}$ , and the coreflections

$$\mathbf{E} \hookrightarrow \mathbf{TL}_0 \hookrightarrow \mathbf{A}, \quad \mathbf{E} \hookrightarrow \mathbf{A}_0 \hookrightarrow \mathbf{N}, \quad \mathbf{A}_c \hookrightarrow \mathbf{Safe},$$

to the categories of labelled structures. Lemma 12.0.2 requires that each functor is associated with a natural transformation relating the events of the image to those originally. In most cases the functors leave the event sets unchanged, which makes the identity natural transformations the evident associates of the adjoint functors and the verification of condition (1) of lemma 12.0.2 a triviality. One exception is the right adjoint of the coreflection from  $\mathbf{TL}_0$  to  $\mathbf{E}$ , dealt with in section 11.2. Another is the functor  $na_0 : \mathbf{N} \rightarrow \mathbf{A}_0$  which has the effect on event sets of reducing them to those events which are reachable. Accordingly, when extending this functor to labelled structures we take the natural transformation associated with  $na_0$  to have components the inclusion of the events of  $na_0(N)$  in those of a net  $N$ . A straightforward application of lemma 12.0.2 lifts the coreflection between asynchronous transition systems and nets to labelled structures. We obtain an adjunction between  $\mathcal{L}(\mathbf{A})$  and  $\mathcal{L}(\mathbf{N})$ , and the coreflections

$$\mathcal{L}(\mathbf{E}) \hookrightarrow \mathcal{L}(\mathbf{TL}_0) \hookrightarrow \mathcal{L}(\mathbf{A}), \quad \mathcal{L}(\mathbf{E}) \hookrightarrow \mathcal{L}(\mathbf{A}_0) \hookrightarrow \mathcal{L}(\mathbf{N}), \quad \mathcal{L}(\mathbf{A}_c) \hookrightarrow \mathcal{L}(\mathbf{Safe}).$$

We can use the adjunctions to relate constructions, and thus semantics, across different categories of labelled structures.

In section 8.3.3 we saw that the reflection between languages and synchronisation trees is paralleled by a reflection between Mazurkiewicz trace languages and labelled event structures. Several independence models are generalisations of transition systems: labelled Petri nets, labelled asynchronous transition systems, transition systems with independence. There is a coreflection  $\mathbf{T} \hookrightarrow \mathbf{TI}$  given by regarding a transition system as having an empty independence relation. However, there are no coreflections from transition systems  $\mathbf{T}$  to the categories of labelled nets or asynchronous transition systems. There are none for the irritating reason that, unlike transition systems, these two models allow more than one transition with the same label between two states. This stops the natural bijection required for the “inclusion” of transition systems to be a left adjoint. A more detailed comparison between interleaving and independence models can suggest new models. See, for example, [Sassone *et al.*, 1993a;



Sassone *et al.*, 1993b] for a classification of models which includes a generalisation of Mazurkiewicz traces, associated with a broad class of pomset languages.

**Remark:** An alternative scheme of labelling is possible for the independence models. Instead of labelling events simply by *sets* of labels, in  $\mathbf{Set}_*$ , we can label by sets together with an independence relation, in the category  $\mathbf{Set}_I$ , respecting the independence of events in the labelling function. We met the category  $\mathbf{Set}_I$  of sets with independence earlier in section 8.3.3 and, as an example of the general construction, the category  $\mathcal{L}_I(\mathbf{E})$  of event structures labelled by sets with independence. The general construction for labelling in  $\mathbf{Set}_I$  proceeds as with  $\mathbf{Set}_*$  and similar general lemmas apply; in particular, the adjunctions/coreflections/reflections between the categories of unlabelled structures lift when labelling by sets with independence. For all the independence models excepts nets the evident projection functors from the categories with labelling sets with independence to  $\mathbf{Set}_I$  are bifibrations—for nets we get just cofibrations. This contrasts with the situation when labelling the independence models by  $\mathbf{Set}_*$ . Then the associated projections only form cofibrations, not fibrations; while the labelled categories do have cartesian liftings of total maps between labelling sets, they do not have cartesian liftings of truly partial maps. A limitation with labelling by sets with independence is that the relabelling construction on processes is determined as a cocartesian lifting only when the relabelling function is a morphism of the category  $\mathbf{Set}_I$ , and so preserves independence. The categories labelling by plain sets are more suitable for the semantics of traditional process algebras. Still, we can imagine process algebras, in which processes have sorts consisting of sets with independence, with relabelling operations only for relabelling functions preserving independence.

## 13 Notes

In this chapter we have surveyed a number of models for concurrency, with special emphasis on the use of category theory in relating the models. We have chosen not to go into any detail on the theories and applications of the individual models. In the following we give some references to such work, and work related to our presentation in general.

*Labelled transition systems* are arguably the most fundamental model within theoretical computer science. An early reference is Keller [Keller, 1976]. In the context of concurrency, they are central to the work on process algebras, like CCS, where processes are typically first given a semantics as labelled transition systems on which behavioural equivalences, like bisimulation, and logics, like Hennessy-Milner logic, are then defined. For one prototypical example of such an approach, see Milner's treatment of CCS

in [Milner, 1989], and for surveys of the many equivalences and logics which have been studied see the papers of van Glabbeek [van Glabbeek, 1990b] and [van Glabbeek, 1993].

Labelled transition systems have been introduced here in their most basic form. Many extensions have been suggested and studied. As a simple example, we have assumed that each transition system has one and only one initial state. A similar theory can be developed with a set of initial states, the interpretation being that initially one and only one of the initial states holds, though it is not determined which (a notable difference is that then the coproduct amounts to just disjoint juxtaposition). More significant extensions include explicit representations of concepts like fair, timed, and probabilistic behaviours. Using labelled transition systems as a model for distributed systems, one often needs to restrict the set of infinite behaviours to those which meet certain progress assumptions for the individual components of the system. Extensions of labelled transition systems dealing with this and other notions of fairness may be found in the works of Manna and Pnueli [Manna and Pnueli, 1991]. Recently, a lot of attention has been paid to extensions taking an explicit account of timing aspects, *e.g.* associating a time measure to each transition, see *e.g.* [Nicollin and Sifakis, 1992]. Some work has also been done on versions of labelled transition systems extended with probability distributions associated with nondeterministic branching, as in [Larsen and Skou, 1989]. For all three types of extensions, generalised theories of equivalences and logics have been developed. Specifications typically take the form of an existing formalism extended to fair, timed, or probabilistic behaviours.

*Synchronisation trees* appeared in Milner's early work of CCS [Milner, 1980]. We use the term in a more general sense, of trees in which arcs are labelled by actions which may be, but are not exclusively, CCS actions.

It is fairly common to see languages, or sets of sequences of states, used to give semantics to parallel processes. The expression *Hoare traces* often turns up in this context, stemming from Hoare's article [Hoare, 1981] though the idea did not originate there, for example appearing in the early work on path expressions [Lauer and Campbell, 1975].

Hoare traces and synchronisation trees represent two extremes in a variety of views on the branching structure of behaviours, often referred to as the linear-time versus branching-time spectrum. In the literature they have been given names like acceptance, refusal, ready, and failure semantics. For a comprehensive survey of these views in terms of models, equivalences, and logics see [van Glabbeek, 1990b; van Glabbeek, 1993]. The models in between Hoare traces and synchronisation trees are typically defined in terms of languages of strings decorated with some branching information. For a thorough treatment of one such model (acceptance trees and testing) see Hennessy's book [Hennessy, 1988].

The "partial simulation" morphisms we define on transition systems

seem to have been discovered several times. They bear a close relationship to bisimulation, as pointed out by several authors, *e.g.* [Joyal *et al.*, 1993]. Their relevance to the operators of process algebras like CCS and CSP was first pointed out by Winskel in [Winskel, 1985]. Because morphisms relate the behaviour of a constructed transition system to that of its components, they also play a role in compositional reasoning (see *e.g.* [Winskel, 1990]).

One omission from our categorical explication of operators is a treatment of hiding, in which certain specified actions are made internal. In the case of languages, such an operation of hiding is achieved by the functor  $\lambda_!$  associated with cocartesian lifting; even when  $\lambda$  is partial, and taken to be undefined on the actions to be hidden, it has cocartesian liftings. But this operation does not seem to capture hiding correctly on the branching structures of transition systems and synchronisation trees. Prefixing might also be expected to play a deeper role categorically than it does at present.

Synchronisation algebras were used largely for the purpose of generality in [Winskel, 1982]. They can be regarded as generalising Milner's monoids of actions [Milner, 1983] by allowing asynchrony between processes (however, here we are on sticky ground, as Milner's monoids are open to different interpretations). A similar idea appeared independently in the work of Aczel, and also Bergstra and Klop [Bergstra and Klop, 1984].

Equivalences between operational and denotational semantics, like the one presented for our process language in section 3, are well known from sequential programming languages, *e.g.* [Winskel, 1993]. Here the operational semantics is given in a syntax-directed way using Plotkin's structural operational semantics, SOS [Plotkin, 1981], and the denotational semantics based on the complete partial order approach of Scott [Scott, 1982]. Plotkin's SOS approach has also been used to give operational semantics for high level process languages with value-passing like Occam (see *e.g.* [Camilleri, 1989]). Many equivalence results between operational and denotational semantics exist for the linear-time versus branching-time spectrum mentioned above, see *e.g.* [Brookes, 1985; Brookes *et al.*; Hennessy, 1988].

An early reference for Mazurkiewicz traces is [Mazurkiewicz, 1977], though the material can also be found in [Mazurkiewicz, 1988]. Mazurkiewicz traces are generally defined a little differently. In particular it is not usual to insist on the *coherence* axiom (referred to as *properness* by Mazurkiewicz [Mazurkiewicz, 1988]) in their definition.

As remarked, Mazurkiewicz traces may be viewed as special kinds of labelled partial orders of events. Labelled partial orders of events appeared earlier in the study of concurrency by Lamport [Lamport, 1978] and Petri [Petri, 1977], and have been advocated under the name of pomsets by Pratt in a series of papers beginning with [Pratt, 1986], and by Grabowski under the name of partial words [Grabowski, 1981]. Note that far from all pomsets can be seen as Mazurkiewicz traces. Consequently Mazurkiewicz trace



languages correspond to special kinds of pomset languages (see [Grabowski, 1981], [Bloom and Kwiatkowska, 1992], and [Sassone *et al.*, 1993b] for some results on their formal relationship). Temporal logics for partially ordered behaviours have been studied by Pinter and Wolper [Pinter and Wolper, 1984] and Katz and Peled [Katz and Peled, 1989].

On the other hand, Mazurkiewicz traces may also be seen as a generalisation of normal strings (with the extra notion of independence between letters), and, following this view, much of the theory of classical formal language theory has been lifted to trace languages. As an example, regular trace languages have been characterised by acceptors (the asynchronous automata of Zielonka [Zielonka, 1989]), algebraically (by Ochmanski [Ochmanski, 1985]), and logically (by Thomas [Thomas, 1990]).

*Event structures* of the kind treated here were introduced by Plotkin and the authors in [Nielsen *et al.*, 1981], and the theory of these and generalised event structures developed by Winskel [Winskel, 1980; Winskel, 1984; Winskel, 1987a; Winskel, 1988b]. Event structures can have a general, and not just a binary conflict, and so can represent precisely the dI-domains of Berry (not just the coherent ones). Through replacing the partial order of causal dependency by an enabling relation, they can represent nondistributive domains. Stable event structures bear the same relation to dI-domains as do information systems to Scott domains. The article in [Winskel, 1987a] gives a reasonable survey, and see [Droste, 1989] for some extensions. The characterisations of the domains of configurations as prime algebraic appear in [Nielsen *et al.*, 1981] and [Winskel, 1980], and the realisation that prime algebraicity amounts to precisely distributivity in [Winskel, 1988b; Winskel, 1983].

The difficulty in defining operations like products and parallel compositions on event structures of the form  $(E, \leq, \#)$  has encouraged the use of more general event structures with which it is easier to give semantics to parallel programming languages, or even languages with higher types (see [Winskel, 1987a; Winskel, 1988b]). Provided the more general event structures have coherent dI-domains as domains of configurations an event structure of the form  $(E, \leq, \#)$  can always be extracted as its complete primes. This line has been followed in [Winskel, 1982; Winskel, 1987a; Boudol, ]. The method is similar to that of using another model like trace languages, asynchronous transition systems, or Petri nets to give a semantics, from which an event-structure semantics is then induced by the coreflection. Event-structure semantics for CCS/TCSP-like languages was made systematic by Winskel in [Winskel, 1982], which exploited a new definition of morphism—that which appears here.

A variation of event structures as models for process languages appears in the “flow event structures” of Boudol and Castellani developed in [Boudol, ; Boudol and Castellani, 1991]; here the problems with the definition of parallel composition are overcome at the expense of an un-

usual treatment of restriction, one where the events to be restricted away are made self-conflicting instead of removed [Castellani and Zhang, 1989]. Other variations include the bundle event structures of Langerak [Langerak, 1991], the families of posets of Rensink [Rensink, 1993], and the event automata of Gunawardena [Gunawardena, 1992] and Pinna and Poigné [Pinna and Poigné, 1992].

Like most of our models, event structures have been equipped with notions of behavioural equivalences (like the history-preserving bisimulation of Rabinovich and Trakhtenbrot [Rabinovich and Trakhtenbrot, 1988]) and logics (for some axiomatizations see the works of Mukund, Thiagarajan [Mukund and Thiagarajan, 1989; Mukund and Thiagarajan, 1990], and Penczek [Penczek, 1988]).

The relationship between event structures and Mazurkiewicz trace languages seems first to have been made explicit by Bednarczyk [Bednarczyk, 1988]. However, the proof of the representation theorem here appears to be new. For an alternative proof see Rozoy and Thiagarajan [Rozoy and Thiagarajan, 1991].

A good reference on *Petri nets* is [Adv, 1987]. The version of Petri nets we describe can be found in the paper [Mazurkiewicz, 1988] of Mazurkiewicz. They are more general than Thiagarajan's elementary net systems [Thiagarajan, 1987] because they allow an event to occur even when there is a condition which is simultaneously a pre- and post-condition. There is a well-known technique known as "complementation" for making a nonsafe net safe. It is notable that this construction comes out of the adjunction between nets and asynchronous transition systems. Our version of Petri nets has been used as a semantic model for process languages in the works of, for example, Olderog [Olderog, 1991].

There are several versions of morphisms on nets in the literature, some more deserving of attention than others. We have examined two. The original definition by Petri [Petri, 1977] seems to have been motivated by graph-theoretic considerations—Petri's morphisms do not respect the behaviour of nets. To some extent the ideas presented here generalise to nets in which events can fire and markings hold with multiplicities, as indicated in [Winskel, 1988a], though at present it is not known how to link up with other models via adjunctions. See also Meseguer and Montanari's study of several definitions of net morphisms [Meseguer and Montanari, 1988].

Categories of Petri nets have been shown to form a model of Girard's linear logic, offering an interpretation of the logical operations of linear logic as operations on nets and of proofs as kinds of simulation morphisms like those here (see [Brown and Gurr, to appear]). Since the morphisms preserve behaviour, the existence of a morphism from one net to another may be interpreted as saying that one net (the implementation) satisfies another (the specification). Recently categories of games have been shown



to be models of linear classical logic [Abramsky and Jagadeesan, 1992; Hyland and Ong, 1993; Lamarche, 1992; Curien, 1993]. The games have the structure of special Petri nets in which the distinction between moves of a player and an opponent is made through one being conditions and the other events (linear negation is caught as reversal of the roles of the players corresponding to swapping the nature of conditions and events). Morphisms are identified with (partial) strategies. As well as providing a refinement of Berry and Curien's sequential algorithms the new categories are suggestive of new paradigms for computation.

*Asynchronous transition systems* are due to Bednarczyk [Bednarczyk, 1988] and Shields [Shields, 1985] who discovered them independently. Bednarczyk's thesis [Bednarczyk, 1988] contains the definition of the category of asynchronous transition systems and the coreflections with event structures and Mazurkiewicz traces. Transition systems with independence are related to the concurrent transition systems of Stark [Stark, 1989]. A related "geometric" approach towards noninterleaving transition systems is taken by Pratt in [Pratt, 1991] and Goubault and Jensen in [Goubault and Jensen, 1992].

As remarked, when presented as transition systems with independence, asynchronous transition systems are amenable to the same techniques (*e.g.* definition by structural operational semantics) as ordinary transition systems. Alternatively, asynchronous transition systems can arise directly through operational semantics, but where instead of just labels, transitions carry more complicated information from which event names and independence can be extracted (see [Boudol and Castellani, 1988; Mukund and Nielsen, 1992; Badouel and Darondeau, 1992] for three examples of this approach). The use of asynchronous transition systems in semantics is often less clumsy than that of nets, which can be extracted afterwards via the adjunction with nets—though sometimes care must be taken to show that the constructions used stay within  $\mathbf{A}_0$ .

The adjunction between asynchronous transition systems and nets is new. It can be viewed as an extension of the adjunction between elementary net systems and elementary transition systems of Nielsen, Rozenberg, and Thiagarajan [Nielsen *et al.*, 1992]. A similar result for general place transition nets and a class of transition systems is presented by Mukund in [Mukund, 1990].

A lot of attention has been paid to noninterleaving semantics in the presence of operators changing the level of atomicity of actions. It turns out that such operators do not admit compositional definitions in the interleaving models, and hence they motivate directly the study of noninterleaving. For examples, see the complementary works of Vogler [Vogler, 1993] and Boudol [Boudol, 1989]. Other examples of equivalences of operationally and denotationally defined noninterleaving semantics have been provided by Boudol and Castellani [Boudol and Castellani, 1991], Degano,

De Nicola, and Montanari [Degano *et al.*, 1988], Gorrieri [Gorrieri, 1992], and Mukund and Nielsen [Mukund and Nielsen, 1992].

Transition systems play an important role in model checking. Interestingly, the extra notion of independence has recently proved to be of value in the search for efficient model checkers—see the works of Wolper [Wolper and Godefroid, 1993].

We have chosen in our treatment not to discuss the various equivalences or logics that one might impose on the models. A good survey of the intimidating range of possible equivalences is given in [van Glabbeek, 1990a] and for noninterleaving models see *e.g.* [Degano *et al.*, 1987]. Some evidence that categorical ideas might help clean up the mess is contained in [Joyal *et al.*, 1993]; there a method is shown for obtaining a generalisation of bisimulation equivalence on categories of models like those here. A central notion in [Joyal *et al.*, 1993] is that of *open map* which, restored to the topos setting where it originated, yields presheaf categories as models, into which synchronisation trees and labelled event structures embed fully and faithfully.

The work on relating models for concurrency has been pursued by others, like Bednarczyk [Bednarczyk, 1988], Rensink [Rensink, 1993], and Kwiatkowska [Kwiatkowska, 1989]. A fuller picture than the one presented here in section 12 has been worked out by Sassone and the authors [Sassone *et al.*, 1993a]. For good introductions to category theory we refer to [Barr and Wells, 1990] and [MacLane, 1971].

Haunting this survey of models for concurrency and their relationship has been the feeling, from time to time, that perhaps the existing models are not quite the right ones, that the lack of existence of certain operations is due to an inadequacy in the models as they are traditionally presented. As our knowledge and experience of what is required of languages and models for parallel computation increases, we will surely be led to richer models and to understand better what structure they should possess. And whatever their form, we should understand how the new models fit with the traditional models studied here.

## Acknowledgements

We are grateful to P. S.Thiagarajan, Bart Jacobs, Nils Klarlund, Peter Knijnenburg, Vladimiro Sassone, and Rob van Glabbeek for helpful suggestions. In particular, Bart Jacobs supplied the short proofs for the appendix on fibred categories. Allan Cheng and Bettina Blaaberg Sørensen are to be thanked for spotting several errors in an earlier draft. Thanks to Uffe Engberg and Madhaven Mukund for their preparation of several of the diagrams.

## References

- [Abramsky and Jagadeesan, 1992] S. Abramsky and R. Jagadeesan. Games and full completeness for multiplicative linear logic. Technical Report DoC 92/24, Imperial College, London, 1992.
- [Adv, 1987] *Advanced course on Petri nets*. LNCS 254, 255, Springer-Verlag, 1987.
- [Badouel and Darondeau, 1992] E. Badouel and P. Darondeau. Structural operational specifications and trace automata. In W.R. Cleaveland, editor, *Concur '92*, pages 302–316. LNCS 630, Springer-Verlag, 1992.
- [Barr and Wells, 1990] M. Barr and C. Wells. *Category Theory for Computer Science*. Prentice Hall, 1990.
- [Bednarczyk, 1988] M. A. Bednarczyk. *Categories of asynchronous systems*. PhD thesis, University of Sussex, 1988.
- [Bénabou, 1985] J. Bénabou. Fibred categories and the foundations of naive category theory. *Journal of Symbolic Logic*, 50:10–37, 1985.
- [Bergstra and Klop, 1984] J. A. Bergstra and J. W. Klop. Process algebra for communication and mutual exclusion. Technical Report CS-R8409, Centrum voor Wiskunde en Informatica, Amsterdam, 1984.
- [Berry, 1979] G. Berry. *Modèles complètement adéquats et stables des  $\lambda$ -calculs typés*. PhD thesis, Université Paris VII, 1979.
- [Bloom and Kwiatkowska, 1992] B. Bloom and M. Kwiatkowska. Trade-offs in true concurrency: Pomsets and Mazurkiewicz traces. In S. Brookes, M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Mathematical Foundations of Programming Semantics*, pages 350–375. LNCS 598, Springer-Verlag, 1992.
- [Boudol, 1989] G. Boudol. Atomic actions. *Bulletin of the European Association for Theoretical Computer Science*, 38:136–144, 1989.
- [Boudol, ] G. Boudol. Flow event structures and flow nets. In I. Guessarian, editor, *Semantics of Systems of Concurrent Processes*, pages 62–95. LNCS 469, Springer-Verlag, .
- [Boudol and Castellani, 1988] G. Boudol and I. Castellani. A non-interleaving semantics for ccs based on proved transitions. *Fundamenta Informaticae*, XI(4):433–452, 1988.
- [Boudol and Castellani, 1991] G. Boudol and I. Castellani. Flow models of distributed computations: three equivalent semantics for CCS. Technical Report 1484, INRIA, 1991. To appear in *Information and Computation*.
- [Brookes, 1983] S. D. Brookes. On the relationship of CCS and CSP. In J. Diaz, editor, *Icalp '83*, pages 83–96. LNCS 154, Springer-Verlag, 1983.
- [Brookes, 1985] S. D. Brookes. On the axiomatic treatment of concurrency. In S. D. Brookes, A. W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency*, pages 1–34. LNCS 197, Springer-Verlag, 1985.

- [Brookes *et al.*, ] S. D. Brookes, A. W. Roscoe, and D. J. Walker. An operational semantics for CSP. Submitted for publication.
- [Brown and Gurr, to appear] C. Brown and D. Gurr. A linear specification language for petri nets. *Mathematical Structures in Computer Science*, to appear.
- [Camilleri, 1989] J. Camilleri. An operational semantics for occam. *International Journal of Parallel Programming*, 18(5), 1989.
- [Castellani and Zhang, 1989] I. Castellani and G. Q. Zhang. Parallel product of event structures. Technical Report 1078, INRIA, 1989.
- [Curien, 1993] P.-L. Curien. *Categorical combinators, sequential algorithms, and functional programming*. Birkhäuser, 1993.
- [Degano *et al.*, 1987] P. Degano, R. De Nicola, and U. Montanari. Observational equivalences for concurrency models. In M. Wirsing, editor, *Formal Description of Programming Concepts – III, IFIP*, pages 105–132. Elsevier Science Publishers B.V., 1987.
- [Degano *et al.*, 1988] P. Degano, R. De Nicola, and U. Montanari. On the consistency of ‘truly concurrent’ operational and denotational semantics, extended abstract. In *IEEE Third Annual Symposium on Logic in Computer Science*, pages 133–141. Computer Society Press, 1988.
- [Droste, 1989] M. Droste. Event structures and domains. *Theoretical Computer Science*, 68:37–47, 1989.
- [van Glabbeek, 1990a] R. J. van Glabbeek. *Comparative concurrency semantics and refinement of actions*. PhD thesis, CWI Amsterdam, 1990.
- [van Glabbeek, 1990b] R. J. van Glabbeek. The linear time - branching time spectrum. In J.C.M. Baeten and Klop J.W., editors, *Concur 90*, pages 278–297. LNCS 458, Springer-Verlag, 1990.
- [van Glabbeek, 1993] R. J. van Glabbeek. The linear time - branching time spectrum II. In E. Best, editor, *Concur 93*, pages 66–81. LNCS 715, Springer-Verlag, 1993.
- [Gorrieri, 1992] R. Gorrieri. A hierarchy of system descriptions via atomic linear refinement. *Fundamenta Informaticae*, 16:289–336, 1992.
- [Goubault and Jensen, 1992] E. Goubault and T. P. Jensen. A homology of higher dimensional automata. In W.R. Cleaveland, editor, *Concur 92*, pages 254–268. LNCS 630, Springer-Verlag, 1992.
- [Grabowski, 1981] J. Grabowski. On partial languages. *Fundamenta Informaticae*, IV(2):427–498, 1981.
- [Grothendieck, 1971] A. Grothendieck. Catégories fibrées et descente. In A. Grothendieck, editor, *Revêtement étales et groupe fondamental, (SGA 1), Exposé VI*. Lecture Notes in Mathematics 224, Springer-Verlag, 1971.
- [Gunawardena, 1992] J. Gunawardena. Causal automata. *Theoretical Computer Science*, 101:265–288, 1992.



- [Hennessy, 1988] M. Hennessy. *Algebraic theory of processes*. MIT Press, 1988.
- [Hoare, 1981] C. A. R. Hoare. A model for communicating sequential processes. Technical Report PRG-22, Programming Research Group, University of Oxford Computing Lab., 1981.
- [Hoare *et al.*, 1984] C. A. R. Hoare, S. D. Brookes, and A. W. Roscoe. A theory of communicating processes. *JACM*, 31(3):560–599, 1984.
- [Hyland and Ong, 1993] J. M. E. Hyland and C.-H. L. Ong. Fair games and full completeness for multiplicative linear logic without the mix-rule. Technical Report, Computer Laboratory, University of Cambridge, 1993.
- [Joyal *et al.*, 1993] A. Joyal, M. Nielsen, and G. Winskel. Bisimulation and open maps. In *Proc. of LICS 93*, pages 418–427, 1993.
- [Kahn and Plotkin, 1979] G. Kahn and G. Plotkin. Structures de données concrètes. Technical Report 336, IRIA-Laboria, 1979.
- [Katz and Peled, 1989] S. Katz and D. Peled. An efficient verification method for parallel and distributed programs. In de Bakker, de Roever, and Rozenberg, editors, *Linear Time, Branching Time and Partial Orders in Logics and Models for Concurrency*, pages 489–507. LNCS 354, Springer-Verlag, 1989.
- [Keller, 1976] R. M. Keller. Formal verification of parallel programs. *CACM*, 19(7):371–384, 1976.
- [Kwiatkowska, 1989] M. Kwiatkowska. *Categories of Asynchronous Systems*. PhD thesis, University of Leicester, 1989.
- [Lamarche, 1992] F. Lamarche. Sequentiality, games and linear logic. In *Proceedings of CLICS Workshop—Part I & II, Aarhus, March 1992*. DAIMI PB 398, Aarhus University, 1992.
- [Lamport, 1978] L. Lamport. Time, clocks and the ordering of events in a distributed system. *CACM*, 21:558–565, 1978.
- [Langerak, 1991] R. Langerak. Bundle event structures: A non-interleaving semantics of LOTOS. Technical Report Memoranda Informatica 91–60, University of Twente, 1991.
- [Larsen and Skou, 1989] K. Larsen and A. Skou. Bisimulation through probabilistic testing. In *Principles of Programming Languages*, pages 344–351, 1989.
- [Lauer and Campbell, 1975] P. Lauer and R. H. Campbell. Formal semantics for a class of high level primitives for coordinating concurrent processes. *Acta Informatica*, 5:297–332, 1975.
- [MacLane, 1971] S. MacLane. *Categories for the Working Mathematician*. Graduate Texts in Mathematics, Springer-Verlag, 1971.
- [Manna and Pnueli, 1991] Z. Manna and A. Pnueli. *The temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1991.



- [Mazurkiewicz, 1977] A. Mazurkiewicz. Concurrent program schemes and their interpretations. Technical Report PB-78, DAIMI, Computer Science Department, University of Aarhus, 1977.
- [Mazurkiewicz, 1988] A. Mazurkiewicz. Basic notions of trace theory. In de Bakker, de Roever, and Rozenberg, editors, *Linear Time, Branching Time and Partial Orders in Logics and Models for Concurrency*, pages 285–363. LNCS 354, Springer-Verlag, 1988.
- [Meseguer and Montanari, 1988] J. Meseguer and U. Montanari. Petri nets are monoids: a new algebraic foundation for net theory. In *Proc. of LICS 88*, pages 155–164, 1988.
- [Milner, 1980] A. R. G. Milner. *Calculus of communicating systems*. LNCS 92, Springer-Verlag, 1980.
- [Milner, 1983] A. R. G. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.
- [Milner, 1989] A. R. G. Milner. *Communication and concurrency*. Prentice Hall, 1989.
- [Mukund and Nielsen, 1992] M. Mukund and M. Nielsen. CCS, locations and asynchronous transition systems. In R. Shyamasundar, editor, *FST & TCS 92*, pages 328–341. LNCS 652, Springer-Verlag, 1992.
- [Mukund and Thiagarajan, 1989] M. Mukund and P. S. Thiagarajan. An axiomatization of event structures. In C.E. Veni Madhavan, editor, *FST & TCS 89*, pages 143–160. LNCS 405, Springer-Verlag, 1989.
- [Mukund, 1990] M. Mukund. A transition system characterization of petri nets. Technical Report TCS-91-2, School of Mathematics, SPIC Science Foundation, 1990.
- [Mukund and Thiagarajan, 1990] M. Mukund and P. S. Thiagarajan. An axiomatization of well branching prime event structures. Technical Report TCS-90-2, School of Mathematics, SPIC Science Foundation, 1990.
- [Nicollin and Sifakis, 1992] X. Nicollin and J. Sifakis. *An overview and synthesis on timed process algebras*. pages 376–398. LNCS 575, Springer-Verlag, 1992.
- [Nielsen *et al.*, 1981] M. Nielsen, G. Plotkin, and G. Winskel. Petri nets, event structures and domains, part 1. *Theoretical Computer Science*, 13:85–108, 1981.
- [Nielsen *et al.*, 1992] M. Nielsen, G. Rozenberg, and P. S. Thiagarajan. Elementary transition systems. *Theoretical Computer Science*, 96:3–33, 1992.
- [Ochmanski, 1985] E. Ochmanski. Regular behaviour of concurrent systems. *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, 27:56–67, 1985.
- [Olderog, 1991] E.-R. Olderog. *Nets, Terms and Formulas. Three views of Concurrent Processes and Their Relationships*. Cambridge University

Press, 1991.

- [Peirce, 1991] B. C. Peirce. *Category theory for computer scientists*. Foundations of Computing Series. The MIT Press, 1991.
- [Penczek, 1988] W. Penczek. A temporal logic for event structures. *Fundamenta Informaticae*, XI:297–326, 1988.
- [Petri, 1977] C. A. Petri. Non-sequential processes. Technical Report ISF-77-05, GMD-ISF, 1977.
- [Pinna and Poigné, 1992] G. M. Pinna and A. Poigné. On the nature of events. In I.M. Havel and V. Koubek, editors, *Mathematical Foundations of Computer Science v*, pages 430–441. LNCS 629, Springer-Verlag, 1992.
- [Pinter and Wolper, 1984] S. Pinter and P. Wolper. A temporal logic for reasoning about partially ordered computations. In *Proc. 3rd ACM PODC*, pages 24–37, 1984.
- [Plotkin, 1981] G. D. Plotkin. Structural operational semantics. Technical Report Lecture Notes, DAIMI FN-19, Computer Science Department, University of Aarhus, 1981. Reprinted 1991.
- [Pratt, 1986] V. R. Pratt. Modelling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, 1986.
- [Pratt, 1991] V. R. Pratt. Modelling concurrency with geometry. In *Proc. 18th Ann. ACM on Principles of Programming Languages*, pages 311–322, 1991.
- [Rabinovich and Trakhtenbrot, 1988] A. Rabinovich and B. A. Trakhtenbrot. Behaviour structure and nets. *Fundamenta Informaticae*, XI(4):357–404, 1988.
- [Rensink, 1993] A. Rensink. *Models and Methods for Action Refinement*. PhD thesis, University of Twente, 1993.
- [Rozoy and Thiagarajan, 1991] B. Rozoy and P. S. Thiagarajan. Event structures and trace monoids. *Theoretical Computer Science*, 91(2):285–313, 1991.
- [Sassone *et al.*, 1993a] V. Sassone, M. Nielsen, and G. Winskel. A classification of models for concurrency. In E. Best, editor, *Concur '93*, pages 82–96. LNCS 715, Springer-Verlag, 1993.
- [Sassone *et al.*, 1993b] V. Sassone, M. Nielsen, and G. Winskel. Deterministic behavioural models for concurrency. In A.M. Borzyszkowski and S. Sokolowski, editors, *MFCS 93*, pages 682–692. LNCS 711, Springer-Verlag, 1993.
- [Scott, 1982] D. A. Scott. Domains for denotational semantics. In M. Nielsen and E.M. Schmidt, editors, *Automata, Languages and Programming*, pages 577–613. LNCS 140, Springer-Verlag, 1982.
- [Shields, 1985] M. W. Shields. Concurrent machines. *Computer Journal*, 28:449–465, 1985.

- [Stark, 1989] E. W. Stark. Concurrent transition systems. *Theoretical Computer Science*, 64:221–269, 1989.
- [Stirling, 1992] C. Stirling. Modal and temporal logics. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, vol. 1. Oxford University Press, 1992.
- [Thiagarajan, 1987] P. S. Thiagarajan. Elementary net systems. In Brauer, Reissig, and Rozenberg, editors, *Petri Nets: Central models and their properties*, pages 26–59. LNCS 254, Springer-Verlag, 1987.
- [Thomas, 1990] W. Thomas. On logical definability of trace languages. In V. Diekert, editor, *Proceedings of a workshop of the ESPRIT Basic Research Action no 3166: Algebraic and Syntactic Methods in Computer Science (ASMICS)*, Kochel am See, Bavaria, FRG, pages 172–182, 1990. Report TUM 19002, Technical University of Munich.
- [Vogler, 1993] W. Vogler. Bisimulation and action refinement. *Theoretical Computer Science*, 114:173–200, 1993.
- [Winskel, 1980] G. Winskel. *Events in Computation*. PhD thesis, University of Edinburgh, 1980.
- [Winskel, 1982] G. Winskel. Event structure semantics of ccs and related languages. In M. Nielsen and E.M. Schmidt, editors, *Icalp '82*, pages 561–576. LNCS 140, Springer-Verlag, 1982. A full version with proofs appears as DAIMI PB-159, Computer Science Department, University of Aarhus, 1983.
- [Winskel, 1983] G. Winskel. A representation of completely distributive algebraic lattices. Technical Report, Computer Science Department, Carnegie-Mellon University, 1983.
- [Winskel, 1984] G. Winskel. Categories of models for concurrency. In S. D. Brookes, W. W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency*, pages 246–267. LNCS 197, Springer-Verlag, 1984.
- [Winskel, 1985] G. Winskel. Synchronisation trees. *Theoretical Computer Science*, 34:33–82, 1985.
- [Winskel, 1987a] G. Winskel. Event structures. In Brauer, Reissig, and Rozenberg, editors, *Petri Nets: Applications and relationships to other models of concurrency*, pages 325–392. LNCS 255, Springer-Verlag, 1987.
- [Winskel, 1987b] G. Winskel. Petri nets, algebras, morphisms and compositionality. *Information and Computation*, 72:197–238, 1987.
- [Winskel, 1988a] G. Winskel. A category of labelled petri nets and compositional proof system. In *Proc. of LICS 88*, pages 142–154, 1988.
- [Winskel, 1988b] G. Winskel. An introduction to event structures. In de Bakker, de Roever, and Rozenberg, editors, *Linear Time, Branching Time and Partial Orders in Logics and Models for Concurrency*, pages 364–397. LNCS 354, Springer-Verlag, 1988.
- [Winskel, 1990] G. Winskel. A compositional proof system on a category

- of labelled transition systems. *Information and Computation*, 87:2–57, 1990.
- [Winskel, 1993] G. Winskel. *The Formal Semantics of Programming Languages. An Introduction*. The MIT Press, 1993.
- [Wolper and Godefroid, 1993] P. Wolper and P. Godefroid. Partial-order methods for temporal verification. In E. Best, editor, *Concur '93*, pages 233–246. LNCS 715, Springer-Verlag, 1993.
- [Zielonka, 1989] W. Zielonka. Safe executions of recognizable trace languages by asynchronous automata. In A. R. Meyer *et al.*, editors, *Proc. Symposium on Logical Foundations of Computer Science, Logic at Botik '89, Pereslavl-Zalessky (USSR)*, pages 278–289. LNCS 363, Springer-Verlag, 1989.

## Appendices

### A A basic category

We shall work with a particular representation of the category of sets with partial functions. Assume that  $X$  and  $Y$  are sets not containing the distinguished symbol  $*$ . Write  $f : X \rightarrow_* Y$  for a function  $f : X \cup \{*\} \rightarrow Y \cup \{*\}$  such that  $f(*) = *$ . When  $f(x) = *$ , for  $x \in X$ , we say  $f(x)$  is *undefined* and otherwise *defined*. We say  $f : X \rightarrow_* Y$  is *total* when  $f(x)$  is defined for all  $x \in X$ . Of course, such total morphisms  $X \rightarrow_* Y$  correspond to the usual total functions  $X \rightarrow Y$ , with which they shall be identified. For the category  $\mathbf{Set}_*$ , we take as objects sets which do not contain  $*$ , and as morphisms functions  $f : X \rightarrow_* Y$ , with the composition of two such functions being the usual composition of total functions (but on sets extended by  $*$ ). Of course,  $\mathbf{Set}_*$  is isomorphic to the category of sets with partial functions, as usually presented.

We remark on some categorical constructions in  $\mathbf{Set}_*$ . A coproduct of  $X$  and  $Y$  in  $\mathbf{Set}_*$  is the disjoint union  $X \uplus Y$  with the obvious injections. A product of  $X$  and  $Y$  in  $\mathbf{Set}_*$  has the form

$$X \times_* Y = \{(x, *) \mid x \in X\} \cup \{(*, y) \mid y \in Y\} \cup \{(x, y) \mid x \in X, y \in Y\}$$

with projections those partial functions to the left and right coordinates.

### B Fibred categories

Our presentation relies on some basic notions from fibred category theory originating in the work of Grothendieck [Grothendieck, 1971], and Bénabou [Bénabou, 1985].<sup>12</sup>

**Definition B.0.1.** Let  $p : \mathbf{X} \rightarrow \mathbf{B}$  be a functor.

---

<sup>12</sup>Our presentation has been improved by incorporating proofs of Bart Jacobs.



A morphism  $f : X \rightarrow X'$  in  $\mathbf{X}$  is said to be *cartesian* with respect to  $p$  if for any morphism  $g : Y \rightarrow X'$  in  $\mathbf{X}$  such that  $p(g) = p(f)$  there is a unique morphism  $h : Y \rightarrow X$  such that  $p(h) = 1_{p(X)}$  and  $f \circ h = g$ . As a diagram:

$$\begin{array}{ccc}
 & Y & \\
 & \downarrow h & \searrow g \\
 X & \xrightarrow{f} & X' \\
 \downarrow p & & \\
 \mathbf{B} & \xrightarrow{p(f)} & p(X')
 \end{array}$$

A cartesian morphism  $f : X \rightarrow X'$  in  $\mathbf{X}$  is said to be a *cartesian lifting* of the morphism  $p(f)$  in  $\mathbf{B}$  with respect to  $X'$ .

Say  $p : \mathbf{X} \rightarrow \mathbf{B}$  is a *fibration* if

- every morphism  $\lambda : B \rightarrow B'$  in  $\mathbf{B}$  has a cartesian lifting with respect to any  $X'$  such that  $p(X') = B'$ , and
- any composition of cartesian morphisms is again cartesian.

A morphism  $f : X \rightarrow X'$  in  $\mathbf{X}$  is said to be *vertical* if  $p(f) = 1_{p(X)}$ .

Often  $p$  is called the *projection*,  $\mathbf{B}$  the *base category*, and each subcategory  $p^{-1}(B)$  of  $\mathbf{X}$ , which is sent to the subcategory consisting of the identity morphism on an object  $B$  of  $\mathbf{B}$ , the *fibre* over  $B$ .

A fibration can also be presented a little differently. A morphism  $f : X \rightarrow X'$  in  $\mathbf{X}$  is said to be *strong cartesian* with respect to a functor  $p : \mathbf{X} \rightarrow \mathbf{B}$  if for any  $g : Y \rightarrow X'$  in  $\mathbf{X}$  and morphism  $\lambda : p(Y) \rightarrow p(X)$  in  $\mathbf{B}$  for which  $p(f) \circ \lambda = p(g)$  there is a unique morphism  $h : Y \rightarrow X$  such that  $p(h) = \lambda$  and  $f \circ h = g$ . It is not hard to show that strong cartesian morphisms compose and that any strong cartesian morphism is cartesian. Moreover, in a fibration any cartesian morphism is strong cartesian (again not hard to show). Hence a fibration can alternatively be defined as a functor  $p : \mathbf{X} \rightarrow \mathbf{B}$  for which each morphism in the base category possesses a strong cartesian lifting (without needing the further requirement that cartesian maps compose).

**Definition B.0.2.** Let  $p : \mathbf{X} \rightarrow \mathbf{B}$  be a functor. It is a *cofibration* if  $p^{op} : \mathbf{X}^{op} \rightarrow \mathbf{B}^{op}$  is a fibration. A morphism  $f : X \rightarrow X'$  in  $\mathbf{X}$  is said to be *cocartesian* with respect to  $p$  if  $f^{op}$  is cartesian in the fibration; the morphism  $f$  is a *cocartesian lifting* of  $p(f)$ . (Call  $f$  strong cocartesian if  $f^{op}$  is strong cartesian.) We say that  $p$  is a *bifibration* if it is both a fibration and cofibration.

Thus a morphism  $f : X \rightarrow X'$  in  $\mathbf{X}$  is cocartesian with respect to  $p$  if for any morphism  $g : X' \rightarrow Y$  in  $\mathbf{X}$  such that  $p(g) = p(f)$  there is a unique morphism  $h : X' \rightarrow Y$  such that  $p(h) = 1_{p(X')}$  and  $h \circ f = g$ . As a diagram:



$$\begin{array}{ccc}
 & & Y' \\
 & \nearrow g & \uparrow h \\
 X & \xrightarrow{f} & X' \\
 \downarrow p & & \downarrow \\
 \mathbf{B} & \xrightarrow{p(f)} & p(X')
 \end{array}$$

For later proofs it is convenient to have a characterisation of (strong) cartesian morphisms. Let  $p : \mathbf{X} \rightarrow \mathbf{B}$  be a functor. For  $X, X'$  in  $\mathbf{X}$  and  $\lambda : p(X) \rightarrow p(X')$  in  $\mathbf{B}$ , write

$$\mathbf{X}_\lambda(X, X') =_{\text{def}} \{f : X \rightarrow X' \mid p(f) = \lambda\}.$$

It is easily verified that:

**Proposition B.0.3.** *A morphism  $f : X \rightarrow X'$  in  $\mathbf{X}$ , with  $p(f) = \lambda$ , is strong cartesian iff for each  $X''$  in  $\mathbf{X}$  and  $\lambda' : p(X'') \rightarrow p(X)$ , the map*

$$(f \circ -) : \mathbf{X}_{\lambda'}(X'', X) \rightarrow \mathbf{X}_{\lambda \circ \lambda'}(X'', X'),$$

*obtained by composing with  $f$ , is an isomorphism (of sets, i.e. a bijection). A morphism  $f : X \rightarrow X'$  in  $\mathbf{X}$ , with  $p(f) = \lambda$ , is cartesian iff for each  $X''$  in  $\mathbf{X}$ , the map*

$$(f \circ -) : \mathbf{X}_{1_{p(X)}}(X'', X) \rightarrow \mathbf{X}_{\lambda \circ 1_{p(X)}}(X'', X'),$$

*obtained by composing with  $f$ , is an isomorphism.*

(Strong) cocartesian morphisms can be characterised similarly.

We are also concerned with functors  $F : \mathbf{X} \rightarrow \mathbf{Y}$  between fibrations  $p : \mathbf{X} \rightarrow \mathbf{B}$  and  $q : \mathbf{Y} \rightarrow \mathbf{B}$ . The functors will preserve the base category in the sense that

$$q \circ F = p.$$

Such functors are said to be *cartesian* when they preserve cartesian morphisms. As the next lemma shows, this property will be automatic for right adjoints of *fibrewise* adjunctions, i.e. those in which the functors preserve the base category and cut down to adjunctions between fibres over common objects in the base category. A dual result holds for left adjoints and cofibrations.

**Definition B.0.4.** Suppose  $p : \mathbf{X} \rightarrow \mathbf{B}$  and  $q : \mathbf{Y} \rightarrow \mathbf{B}$  and that functors  $F : \mathbf{X} \rightarrow \mathbf{Y}$  and  $G : \mathbf{Y} \rightarrow \mathbf{X}$  form an adjunction with  $F$  left adjoint to  $G$ . The adjunction is said to be *fibrewise*, with respect to  $p$  and  $q$ , iff  $q \circ F = p$  and  $p \circ G = q$  and each component of the counit  $\epsilon_Y : FG(Y) \rightarrow Y$  is

vertical, for  $Y \in \mathbf{Y}$ , i.e.  $q(\epsilon_Y) = 1_{q(Y)}$  (or equivalently, components of the unit are vertical).

**Lemma B.0.5.** *Suppose  $p : \mathbf{X} \rightarrow \mathbf{B}$  and  $q : \mathbf{Y} \rightarrow \mathbf{B}$  and that functors  $F : \mathbf{X} \rightarrow \mathbf{Y}$  and  $G : \mathbf{Y} \rightarrow \mathbf{X}$  form a fibrewise adjunction with  $F$  left adjoint to  $G$ . Then the right adjoint  $G$  preserves strong cartesian morphisms, and cartesian morphisms.*

**Proof.** Assume  $g : Y \rightarrow Y'$  in  $\mathbf{Y}$  is strong cartesian over  $\beta$  in  $\mathbf{B}$ .

Naturality of the adjunction yields that the diagram

$$\begin{array}{ccc} \mathbf{X}(X, G(Y)) & \xrightarrow{\theta} & \mathbf{Y}(F(X), Y) \\ (G(g) \circ -) \downarrow & & \downarrow (g \circ -) \\ \mathbf{X}(X, G(Y')) & \xrightarrow{\theta'} & \mathbf{Y}(F(X), Y') \end{array}$$

commutes, where  $X \in \mathbf{X}$  and  $\theta, \theta'$  are components of the natural isomorphisms of the adjunction. Because the adjunction is fibrewise,  $p(-) = q \circ \theta(-)$ .

Letting  $X \in \mathbf{X}$  and  $\alpha : p(X) \rightarrow pG(Y)$ , we observe the following chain of isomorphisms

$$\begin{aligned} \mathbf{X}_\alpha(X, G(Y)) & \xrightarrow{\theta} \mathbf{Y}_\alpha(F(X), Y) \\ & \xrightarrow{(g \circ -)} \mathbf{Y}_{\beta \circ \alpha}(F(X), Y') \text{ as } g \text{ is strong cartesian} \\ & \xrightarrow{\theta'^{-1}} \mathbf{X}_{\beta \circ \alpha}(X, G(Y')) \end{aligned}$$

whose composition,  $\theta'^{-1}(g \circ \theta(-))$ , equals  $(G(g) \circ -)$  from the commuting diagram above. Hence  $G(g)$  is strong cartesian by proposition B.0.3.

The proof in the case of just cartesian morphisms is got by specialising the morphism  $\alpha$  above to the identity. ■

Note that lemma B.0.5 does not state that the left adjoint  $F$  preserves cartesian morphisms. Nor does it entail that the right adjoint  $G$  preserves cocartesian morphisms, and these are not true in general. For instance, they do not hold for the coreflection between synchronisation trees and transition systems.

In the case where the adjunctions form coreflections (or reflections) there are further useful results. The left adjoint of a coreflection (recall that this is an adjunction in which the unit is a natural isomorphism) is automatically full and faithful, as is the right adjoint of a reflection, and thus the following lemma applies to these functors in a fibrewise adjunction:

**Lemma B.0.6.** *Suppose  $p : \mathbf{X} \rightarrow \mathbf{B}$  and  $q : \mathbf{Y} \rightarrow \mathbf{B}$  and that functor  $F : \mathbf{X} \rightarrow \mathbf{Y}$  is full and faithful with  $q \circ F = p$ . Then  $F$  reflects (strong) (co) cartesian morphisms.*

**Proof.** The conditions on  $F$  imply that  $F$  restricts to an isomorphism

$$\mathbf{X}_\gamma(X, X') \xrightarrow{F} \mathbf{Y}_\gamma(F(X), F(X'))$$

for any  $X, X' \in \mathbf{X}$  and  $\gamma : p(X) \rightarrow p(X')$  in  $\mathbf{B}$ .

Let  $X'' \in \mathbf{X}$  and  $\alpha : p(X'') \rightarrow p(X)$  in  $\mathbf{B}$ . The chain of isomorphisms

$$\begin{aligned} \mathbf{X}_\alpha(X'', X) &\xrightarrow{F} \mathbf{Y}_\alpha(F(X''), F(X)) \\ &\xrightarrow{(F(f) \circ -)} \mathbf{Y}_{\beta \circ \alpha}(F(X''), F(X')) \\ &\quad \cong \text{as } F(f) \text{ is strong cartesian,} \\ &\xrightarrow{F^{-1}} \mathbf{X}_{\beta \circ \alpha}(X'', X') \end{aligned}$$

composes to  $F^{-1}(F(f) \circ F(-)) = (f \circ -)$ . Hence  $f$  is strong cartesian by proposition B.0.3.

The proof for just cartesian morphisms follows by specialising  $\alpha$  to the identity. The proof for cocartesian liftings is similar.  $\blacksquare$

## C Operational semantics—proofs

Here we provide the proofs required in showing the equivalence between the denotational and operational semantics of the process language in terms of transition systems of section 3.2 (part (a)) and, by a slightly more general argument, section 11.3 (part (b)). Both parts rely on acyclicity of the transition relation got via the operational semantics.

**Lemma C.0.1 (Lemma 3.2.2).** *For any closed tagged term  $t$ , the transition system  $\mathcal{Op}(t)$  is acyclic.*

**Proof.** We show this by mapping tagged terms  $t$  to  $|t|$  in a strict order  $<$  (an irreflexive, transitive relation) in such a way that

$$t \xrightarrow{a} u \text{ \& } a \neq * \Rightarrow |t| < |u|. \quad (1)$$

It then follows that  $\rightarrow^+$  is irreflexive.

Define  $< \subseteq \omega \times \omega$  by taking  $(m, n) < (m', n') \Leftrightarrow m < m'$  or  $(m = m' \text{ \& } n > n')$ . (In other words  $<$  is the lexicographic combination of  $<$  and  $>$  on integers.) The relation  $<$  is a strict order. For  $t$  a closed tagged term, define

$$|t| = (\text{tag}(t), \text{size}(t))$$

where the functions  $\text{tag}$  and  $\text{size}$  are defined by the following structural inductions:

$$\begin{aligned}
\text{tag}(\text{nil}) &= \text{tag}(x) = 0 \\
\text{tag}(\text{at}) &= \text{tag}(t) \\
\text{tag}(t_0 \oplus t_1) &= \min(\text{tag}(t_0), \text{tag}(t_1)) \\
\text{tag}(t_0 \times t_1) &= \text{tag}(t_0) + \text{tag}(t_1) \\
\text{tag}(t \upharpoonright \Lambda) &= \text{tag}(t\{\Xi\}) = \text{tag}(t) \\
\text{tag}(\text{rec } x.t) &= \text{tag}(t) \\
\text{tag}((l, t)) &= 1 + \text{tag}(t) \\
\text{size}(\text{nil}) &= \text{size}(x) = 0 \\
\text{size}(\text{at}) &= 1 + \text{size}(t) \\
\text{size}(t_0 \oplus t_1) &= 1 + \text{size}(t_0) + \text{size}(t_1) \\
\text{size}(t_0 \times t_1) &= 1 + \text{size}(t_0) + \text{size}(t_1) \\
\text{size}(t \upharpoonright \Lambda) &= \text{size}(t\{\Xi\}) = 1 + \text{size}(t) \\
\text{size}(\text{rec } x.t) &= 1 + \text{size}(t) \\
\text{size}((n, t)) &= 1 + \text{size}(t).
\end{aligned}$$

The rules for operational semantics can be shown to preserve property (1) above, which hence holds for all derivable transitions. For example, considering the rule for recursion, assume

$$|t[\text{rec } x.t/x]| < |t'|$$

holds for the transition  $t[\text{rec } x.t/x] \xrightarrow{a} t'$  in its premise if  $a \neq *$ . It follows that

$$\text{tag}(t[\text{rec } x.t/x]) \leq \text{tag}(t').$$

Clearly  $\text{tag}(t) \leq \text{tag}(t[\text{rec } x.t/x])$  so

$$\text{tag}(\text{rec } x.t) = \text{tag}(t) < 1 + \text{tag}(t') = \text{tag}((2, t')).$$

Hence

$$|\text{rec } x.t| < |(2, t')|$$

holds for the transition  $\text{rec } x.t \xrightarrow{a} (2, t')$ . ■

### (a). Uniqueness for guarded recursions in **T**

We prove lemma 3.0.2, showing that solutions to guarded recursions are unique to within isomorphism. The proof rests on the definition of a family of functors  $(-)^{(k)}$ , for  $k \in \omega$ , “projecting” a transition system to the transition system consisting of that part reachable within  $k$  steps.

**Lemma C.0.2.** *Suppose  $T_{n,m}$  are transition systems for  $n, m \in \omega$  with the property that*

$$T_{n,m} \trianglelefteq T_{n',m'} \text{ when } n \leq n' \text{ and } m \leq m'.$$

*Then the set  $\{T_{n,m} \mid n, m \in \omega\}$  has a least upper bound*

$$\bigcup_{n,m \in \omega} T_{n,m} = \bigcup_{n \in \omega} \left( \bigcup_{m \in \omega} T_{n,m} \right) = \bigcup_{m \in \omega} \left( \bigcup_{n \in \omega} T_{n,m} \right) = \bigcup_{n \in \omega} T_{n,n}.$$

**Proposition C.0.3.**

1. For each  $k \in \omega$ , the operation  $(-)^{(k)}$  is a functor on the category  $\mathbf{T}$  of transition systems; it restricts to an endofunctor on the subcategory where morphisms are label preserving.
2. Let  $T$  be a transition system. Then  $T^{(k)} \sqsubseteq T$ , for  $k \in \omega$ . If  $k \leq l$  then  $T^{(k)} \sqsubseteq T^{(l)}$ . For  $k, l \in \omega$ ,  $(T^{(k)})^{(l)} = T^{\min(k,l)}$ . Recalling the operation  $\mathcal{R}$  of section 3, taking the reachable part of a transition system, we have

$$\mathcal{R}(T) = \bigcup_{k \in \omega} T^{(k)}.$$

3. The operations  $(-)^{(k)}$ , for  $k \in \omega$ , and  $\mathcal{R}$  are continuous with respect to  $\sqsubseteq$ .

**Proof.** (1) As morphisms preserve or collapse transitions, it follows that a morphism  $f : T_0 \rightarrow T_1$  restricts to a morphism  $f^{(k)} : T_0^{(k)} \rightarrow T_1^{(k)}$ . The operation  $(-)^{(k)}$  clearly preserves identities and composition. It is easily checked that these facts also hold when restricting attention to label-preserving morphisms.

(2) is obvious.

(3) From the definition of  $(-)^{(k)}$ , for  $k \in \omega$ , it is easily seen that it is continuous. To show  $\mathcal{R}$  is continuous suppose

$$T_0 \sqsubseteq \dots \sqsubseteq T_n \sqsubseteq \dots$$

is a chain of transition systems. Then

$$\begin{aligned} \mathcal{R}(\bigcup_n T_n) &= \bigcup_k (\bigcup_n T_n)^{(k)} && \text{by the definition of } \mathcal{R} \\ &= \bigcup_k \bigcup_n T_n^{(k)} && \text{by continuity of } (-)^{(k)} \\ &= \bigcup_n \bigcup_k T_n^{(k)} && \text{by lemma C.0.2} \\ &= \bigcup_n \mathcal{R}(T_n) && \text{by the definition of } \mathcal{R}. \end{aligned}$$

■

Say an operation  $F$  on transition systems is definable if it acts on  $T$  so that

$$F(T) = \mathbf{T}[[t]]\rho[T/x]$$

for some choice of process term  $t$  and variable  $x$ . Any operation  $F$  definable in the process language is  $\sqsubseteq$ -monotonic and continuous and has the property that



$$(F(T))^{(k)} = (F(T^{(k)}))^{(k)}. \quad (1)$$

This follows by structural induction from facts such as

$$(T \times U)^{(k)} = (T^{(k)} \times U^{(k)})^{(k)}$$

about the basic operations. A prefixing operation  $a(-)$  has the stronger property that, for  $k > 0$ ,

$$(a(T))^{(k)} = (a(T^{(k-1)}))^{(k)}.$$

It follows that an operation  $F$  defined by a guarded recursion satisfies

$$(F(T))^{(k)} = (F(T^{(k-1)}))^{(k)} \quad (2)$$

for  $k > 0$ . All the operations extend to functors with respect to label-preserving morphisms on transition systems. The facts above extend to morphisms. A functor  $F$  defined by a guarded recursion has the property that

$$(F(f))^{(k)} = (F(f^{(k-1)}))^{(k)} \quad (3)$$

for  $k > 0$ , on label-preserving morphisms  $f$ .

**Definition C.0.4.** We will call a functor  $F$  on the category of transition systems with label-preserving morphisms *guarded* when it satisfies (2) and (3) above.

Now we can complete the proof of lemma 3.0.2:

**Lemma C.0.5 (Lemma 3.0.2).** *If  $F$  is defined from a guarded recursion we have*

$$T \cong \mathcal{R} \circ F(T) \Rightarrow T \cong \mathcal{R}(\text{fix}(F)).$$

(Here, and in the proof below, morphisms are understood to be in the category with label-preserving morphisms. In particular, the isomorphisms above are label preserving.)

**Proof.** Any functor, on the category of transition systems with label-preserving morphisms, definable in the process language is  $\trianglelefteq$ -continuous. In particular, we remark that such a guarded  $F$  has the property that

$$(\text{fix}(F))^{(n)} = (F^k(I))^{(n)} \quad (4)$$

for all  $k, n \in \omega$  for which  $k \geq n$ . This is obviously so for  $n = 0$  when  $(\text{fix}(F))^0 = I = (F^k(I))^{(0)}$ , where  $I$  is the transition system consisting of a single initial state. Assume (4) as an induction hypothesis. We deduce, assuming  $k \geq n + 1$ , that

$$\begin{aligned}
(F^k(I))^{(n+1)} &= (F(F^{k-1}(I)))^{(n+1)} \\
&= (F((F^{k-1}(I))^{(n)}))^{(n+1)} && \text{as } F \text{ is guarded} \\
&= (F((fix(F))^{(n)}))^{(n+1)} && \text{from the induction hypothesis} \\
&&& \text{as } k-1 \geq n \\
&= (F(fix(F)))^{(n+1)} && \text{as } F \text{ is guarded} \\
&= (fix(F))^{(n+1)}.
\end{aligned}$$

We conclude that (4) holds for general  $n \in \omega$ , with  $k \geq n$ , by induction.

With the help of an observation we can simplify the proof notationally. For an operation  $F$  definable in the process language

$$\mathcal{R} \circ F = \mathcal{R} \circ F \circ \mathcal{R}. \quad (5)$$

To see this, for an arbitrary transition system  $T$ , reason that

$$\begin{aligned}
\mathcal{R} \circ F \circ \mathcal{R}(T) &= \mathcal{R} \circ F(\bigcup_k T^{(k)}) && \text{by definition of } \mathcal{R} \\
&= \bigcup_k \mathcal{R} \circ F(T^{(k)}) && \text{by continuity of } \mathcal{R} \text{ and } F \\
&= \bigcup_k \bigcup_n (F(T^{(k)}))^{(n)} && \text{by definition of } \mathcal{R} \\
&= \bigcup_k (F(T^{(k)}))^{(k)} && \text{by lemma C.0.2} \\
&= \bigcup_k (F(T))^{(k)} && \text{by (1)} \\
&= \mathcal{R} \circ F(T) && \text{by definition of } \mathcal{R}.
\end{aligned}$$

It follows that

$$\begin{aligned}
\mathcal{R}(fix(F)) &= \mathcal{R}(\bigcup_n F^n(I)) \\
&= \bigcup_n \mathcal{R} F^n(I) \\
&= \bigcup_n (\mathcal{R} \circ F)^n \mathcal{R}(I) && \text{by repeated use of (5)} \\
&= \bigcup_n (\mathcal{R} \circ F)^n(I) \\
&= fix(\mathcal{R} \circ F).
\end{aligned}$$

Hence, writing  $G = \mathcal{R} \circ F$ , we can restate the goal of our proof (in the statement of the theorem) as

$$T \cong G(T) \Rightarrow T \cong fix(G) \quad (\dagger)$$

where we have  $G$  is  $\leq$ -continuous and guarded (*i.e.* satisfies (2) and (3)), because these properties are assumed of  $F$  and inherited by  $G$ .

To prove  $(\dagger)$ , assume  $T \cong G(T)$  and let

$$\theta : G(T) \cong T$$

name the isomorphism. It is also convenient to let  $u$  be the unique morphism

$$u : I \rightarrow T.$$

By induction on  $n$  we show that

$$(G^n(u))^{(n)} : (G^n(I))^{(n)} \rightarrow (G^n(T))^{(n)}$$

is an isomorphism. The basis case amounts to showing  $u^{(0)} : I^{(0)} \rightarrow T^{(0)}$  is an isomorphism which is trivially so as each transition system consists only of a single initial state. Notice that

$$\begin{aligned} (G^{n+1}(u))^{(n+1)} &= (G(G^n(u)))^{(n+1)} \\ &= (G((G^n(u))^{(n)}))^{(n+1)}, \end{aligned}$$

because  $G$  is guarded. From the fact that  $G$  and  $(-)^{(n+1)}$  are functors and so preserve isomorphism, we now see that  $(G^{n+1}(u))^{(n+1)}$  being an isomorphism follows inductively from  $(G^n(u))^{(n)}$  being an isomorphism.

Let  $\theta_n$  be the isomorphism

$$(\theta \circ G(\theta) \circ \dots \circ G^{(n-1)}(\theta)) : G^n(T) \rightarrow T$$

for  $n \in \omega$ . The fact that  $(-)^{(n)}$  is a functor ensures  $\theta_n^{(n)}$  is also an isomorphism

$$G^n(T)^{(n)} \rightarrow T^{(n)},$$

for  $n \in \omega$ . As remarked in (4) above,  $(fix(G))^{(n)} = (G^n(I))^{(n)}$ . Thus we obtain isomorphisms

$$\phi_n =_{def} (\theta_n \circ (G^n(u)))^{(n)} : (fix(G))^{(n)} \rightarrow T^{(n)}$$

for  $n \in \omega$ . These are consistent in the sense that

$$\phi_n = \phi_{n+1}^{(n)},$$

for  $n \in \omega$ , as we will now show.

Let  $j$  be the monomorphism associated with  $I \trianglelefteq G(I)$ . Applying  $G^n$  followed by  $(-)^{(n)}$ , we obtain an inclusion morphism

$$(G^n(j))^{(n)} : (G^n(I))^{(n)} \rightarrow (G^{n+1}(I))^{(n)}.$$

But now by (4),

$$(G^n(j))^{(n)} : (fix(G))^{(n)} \rightarrow (fix(G))^{(n)}$$

which being an inclusion morphism must be the identity, *i.e.*

$$(G^n(j))^{(n)} = 1_{(fix(G))^{(n)}}.$$

Certainly we have

$$u = \theta \circ G(u) \circ j,$$

which may be depicted by the following commuting diagram:

$$\begin{array}{ccc} I & \xrightarrow{u} & T \\ j \downarrow & & \uparrow \theta \\ G(I) & \xrightarrow{G(u)} & G(T) \end{array}$$

Hence, for  $n \in \omega$ , applying the functors  $G^n$  and  $(-)^{(n)}$ , we obtain

$$\begin{aligned} (G^n(u))^{(n)} &= (G^n(\theta))^{(n)} \circ (G^{n+1}(u))^{(n)} \circ (G^n(j))^{(n)} \\ &= (G^n(\theta))^{(n)} \circ (G^{n+1}(u))^{(n)}. \end{aligned}$$

Thus

$$\begin{aligned} \phi_n &= (\theta_n \circ G^n(u))^{(n)} \\ &= (\theta \circ \dots \circ G^{(n-1)}(\theta))^{(n)} \circ (G^n(\theta))^{(n)} \circ (G^{n+1}(u))^{(n)} \\ &= ((\theta_{n+1} \circ G^{n+1}(u))^{(n+1)})^{(n)} \text{ by proposition C.0.3(2)} \\ &= \phi_{n+1}^{(n)}. \end{aligned}$$

It follows that

$$\phi = \bigcup_{n \in \omega} \phi_n$$

is an isomorphism  $\text{fix}(G) \rightarrow T$ . ■

## (b). Semantics in **TI**

This part of the appendix is dedicated to showing the equivalence between the operational and denotational semantics of **Proc** in terms of transition systems with independence:

**Theorem C.0.6 (Theorem 11.3.4).** *Let  $t$  be a closed process term. Then, for an arbitrary environment  $\rho$ ,*

$$\mathcal{Op}(t) \cong \mathcal{R}(\mathbf{TI}[\![t]\!]\rho),$$

*a label-preserving isomorphism of transition systems with independence.*

The proof is very like that of theorem 3.2.3. However, in carrying out the proof we move to more general structures than transition systems with independence, to pre-structures of the same form but which do not necessarily satisfy the axioms required of a transition system with independence. This is because *a priori* we do not know that  $\mathcal{Op}(t)$  is a transition system with independence for  $t$  a closed term.

**Definition C.0.7.** Define **pTI** to be the category consisting of

- objects  $(S, i, L, Tran, I)$  where  $(S, i, L, Tran)$  is a transition system and  $I \subseteq Tran \times Tran$  (the independence relation),
- morphisms are morphisms between the underlying transition systems which preserve independence, *i.e.* a morphism  $(\sigma, \lambda) : T \rightarrow T'$  should satisfy

If  $(s, a, s')$  and  $(u, b, u')$  are independent transitions of  $T$  and  $\lambda(a)$  and  $\lambda(b)$  are both defined, then  $(\sigma(s), \lambda(a), \sigma(s'))$  and  $(\sigma(u), \lambda(b), \sigma(u'))$  are independent transitions of  $T'$ .

Composition is inherited from that in **T**.

It is clear that **TI** is a full subcategory of **pTI** and the two categories share many constructions. The same definitions of the process language operations, given in section 11.3.1 for **TI**, serve also for **pTI**. In particular, the definition of  $\sqsubseteq$  can be lifted to give a least-fixed-point semantics of recursion. All told, we can give a denotational semantics of terms in **Proc** as objects of **pTI**; for a term  $t$  of **Proc**, with respect to an environment  $\rho$  from process variables to objects of **pTI** we obtain a denotation

$$\mathbf{pTI}[\![t]\!]\rho,$$

an object in **pTI**. When the environment  $\rho$  yields objects in the subcategory **TI** for any free variable of  $t$ , the two denotations,  $\mathbf{pTI}[\![t]\!]\rho$  and  $\mathbf{TI}[\![t]\!]\rho$ , coincide.

As in (a), a key idea is that of an endofunctor  $(-)^{(k)}$  on **pTI** “projecting” an object to that part of it which is reachable within  $k$  steps, taking  $T$  to  $T^{(k)} \sqsubseteq T$ . In detail:

**Definition C.0.8.** Let  $T = (S, i, L, Tran, I)$  be in **pTI**. Define  $T^{(k)}$  to be  $(S', i, L', Tran', I')$  where  $S'$  is the subset of states  $S$  reachable by  $k$  or less transitions from  $i$ ,  $Tran'$  is the subset of transitions  $Tran$  which are reachable by  $k$  or less transitions,  $L'$  is the subset of labels  $a \in L$  for which there is a transition  $(s, a, s') \in Tran'$ , and  $I' = I \cap (Tran' \times Tran')$ .

Let  $f = (\sigma, \lambda) : T_0 \rightarrow T_1$  be a morphism of **pTI**. Define  $f^{(k)}$ , for  $k \in \omega$ , to be  $(\sigma', \lambda')$  where  $\sigma'$  is the restriction of  $\sigma$  to the states of  $T_0^{(k)}$  and  $\lambda'$  is the restriction of  $\lambda$  to the labels of  $T_0^{(k)}$ .

This provides the appropriate definition of the family of functors  $(-)^{(k)} : \mathbf{pTI} \rightarrow \mathbf{pTI}$  with which to generalise the argument of part (a). (This really is a generalisation once we regard an ordinary transition system as having an empty independence relation.)

Say an operation  $F$  on objects in **pTI** is definable if it acts on  $T$  so that

$$F(T) = \mathbf{pTI}[\![t]\!]\rho[T/x]$$



for some choice of process term  $t$  and variable  $x$ . Any such definable operation is  $\leq$ -continuous and can be extended to an endofunctor on the subcategory of  $\mathbf{pTI}$  in which morphisms are label preserving. It will also satisfy

$$(F(T))^{(k)} = (F(T^{(k)}))^{(k)}, \text{ for } k \in \omega,$$

—just as in part (a). As there,

$$(a(T))^{(k)} = (a(T^{(k-1)}))^{(k)}$$

so that any operation  $F$  defined by a term in which the variable is guarded satisfies

$$(F(T))^{(k)} = (F(T^{(k-1)}))^{(k)},$$

for  $k > 0$ . Given  $f : T \rightarrow T'$  a morphism in  $\mathbf{TI}$ , for  $k \in \omega$ , the morphism  $f^{(k)} : T^{(k)} \rightarrow T'^{(k)}$  is the restriction of  $f$  to the states and labels of  $T^{(k)}$ . It follows that a functor  $F$  definable by a process term satisfies

$$(F(f))^{(k)} = (F(f^{(k)}))^{(k)},$$

on label-preserving morphisms  $f$ , for  $k \in \omega$ , and when associated with a guarded variable,

$$(F(f))^{(k)} = (F(f^{(k-1)}))^{(k)}.$$

We can now proceed as in part (a), reading “ $\mathbf{pTI}$ ” and “object of  $\mathbf{pTI}$ ” instead of “ $\mathbf{T}$ ” and “transition system”; the proofs of proposition C.0.3 and lemma 3.0.2 are sufficiently abstract to apply equally well in the more general situation of  $\mathbf{pTI}$ . This furnishes a proof of the uniqueness property of guarded recursions in  $\mathbf{pTI}$ :

**Lemma C.0.9.** *Suppose the variable  $x$  is guarded in  $t$ . Let  $T$  be an object of  $\mathbf{pTI}$  and  $\rho$  be an environment in  $\mathbf{pTI}$ . If  $T \cong \mathcal{R}(\mathbf{pTI}[[t]\rho[T/x]])$ , a label-preserving isomorphism, then  $T \cong \mathcal{R}(\mathbf{pTI}[\text{rec } x.t]\rho)$ , a label-preserving isomorphism.*

The next stage is to show:

**Theorem C.0.10.** *Let  $t$  be a closed process term. Then, for an arbitrary environment  $\rho$  in  $\mathbf{pTI}$ ,*

$$\mathcal{O}p(t) \cong \mathcal{R}(\mathbf{pTI}[[t]\rho]),$$

*a label-preserving isomorphism.*

**Proof.** The proof is by structural induction on terms  $t$ , with free variables  $\bar{x}$ , that for all closed terms  $\bar{s}$  chosen as substitutions for the variables  $\bar{x}$ ,

$$\mathcal{O}p(t[\bar{s}/\bar{x}]) \cong \mathcal{R}(\mathbf{pTI}[[t]\rho[\overline{\mathcal{O}p(\bar{s})}/\bar{x}]]),$$

a label-preserving isomorphism. The operational rules have been chosen to make the cases of the induction straightforward for all but that of recursive processes—recourse is made to lemma 3.2.2 expressing acyclicity of the transition relation got operationally. The verification of the case of recursion proceeds as in the proof of theorem 3.2.3, relying on the uniqueness lemma C.0.9 above. ■

We now finally obtain theorem 11.3.4 as a corollary, because, as remarked, the denotations  $\mathbf{pTI}[[t]]\rho$  and  $\mathbf{TI}[[t]]\rho$  coincide when  $t$  is closed.

# Concrete process algebra

J.C.M. Baeten<sup>1</sup> and C. Verhoef<sup>1</sup>

---

## Contents

1	Introduction . . . . .	150
2	Concrete sequential processes . . . . .	152
	2.1 Introduction . . . . .	152
	2.2 Basic process algebra . . . . .	152
	2.3 Recursion in BPA . . . . .	167
	2.4 Projection in BPA . . . . .	169
	2.5 Deadlock . . . . .	181
	2.6 Empty process . . . . .	184
	2.7 Renaming in BPA . . . . .	190
	2.8 The state operator . . . . .	197
	2.9 The extended state operator . . . . .	201
	2.10 The priority operator . . . . .	204
	2.11 Basic process algebra with iteration . . . . .	217
	2.12 Basic process algebra with discrete relative time . . . . .	220
	2.13 Basic process algebra with other features . . . . .	224
	2.14 Decidability and expressiveness results in BPA . . . . .	226
3	Concrete concurrent processes . . . . .	228
	3.1 Introduction . . . . .	229
	3.2 Syntax and semantics of parallel processes . . . . .	229
	3.3 Extensions of PA . . . . .	235
	3.4 Extensions of $PA_\delta$ . . . . .	241
	3.5 Syntax and semantics of communicating processes . . . . .	244
	3.6 Extensions of ACP . . . . .	251
	3.7 Decidability and expressiveness results in ACP . . . . .	255
4	Further reading . . . . .	260

---

<sup>1</sup>J. C. M. Baeten received partial support and C. Verhoef received full support from ESPRIT basic research action 7166, CONCUR2.

Algebra may be considered, in its most general form, as *the science which treats of the combinations of arbitrary signs and symbols by means of defined though arbitrary laws ...* [Peacock, 1830].

## 1 Introduction

Concurrency theory is the branch of computer science that studies the mathematics of concurrent or distributed systems. In concurrency theory, the design of such mathematics is studied and issues concerning the specification and verification of such systems are analysed. Often, a concurrent system is called a process. In order to analyse a large and complex process it is desirable to be able to describe it in terms of smaller and simpler processes. Thus, it seems natural to have some simple processes—the ones that are not subject to further investigation—and operators on them to compose larger ones thus resulting in an algebraic structure. In order to reason about large processes it is often useful to have a set of basic identities between processes at one's disposal. The most relevant identities among them are normally called axioms. The axiomatic and algebraic point of view on concurrency theory is widely known as process algebra.

The most well-known algebraic concurrency theories are the ones known by the acronyms CCS, CSP, and ACP. CCS is the Calculus of Communicating Systems of [Milner, 1980]. Theoretical CSP originates from [Brookes *et al.*, 1984]; the acronym CSP stands for Communicating Sequential Processes. ACP is the Algebra of Communicating Processes; the original reference to ACP is [Bergstra and Klop, 1984a]; we note that recently the full version of [Bergstra and Klop, 1984a] has appeared; see [Bergstra and Klop, 1995]. Of these three, (theoretical) CSP is the most abstract (it identifies more processes than the other two), and tends in the direction of a specification language. The other two, CCS and ACP, are based on the same notion of equivalence (bisimulation equivalence), and are more operationally oriented than CSP. They tend in the direction of a programming language. Of the two, CCS has links to logic and  $\lambda$ -calculus, and ACP may be best characterized as a purely algebraical approach.

In this survey we focus on concrete process algebra. Concrete process algebra is the area of algebraic concurrency theory that does not incorporate a notion called abstraction. Abstraction is the ability to hide information, to abstract information away. The reason that we refrain from incorporating this important issue is that concrete process algebra is already such a large part of the theory that it justifies its own survey. Moreover, it is more and more recognized that for the understanding of issues in large languages it is often convenient first to study such issues in a basic language, a language with less features. For instance, some decidability results in process algebra are obtained in this way. Other examples of such basic

languages are Milner's basic CCS [Milner, 1980], BCCSP [Glabbeek, 1990], ASTP [Nicollin and Sifakis, 1994], TCCS<sub>0</sub> [Moller and Tofts, 1990], BPP [Christensen *et al.*, 1993], and the pair BPA/PA [Bergstra and Klop, 1982]. The results that may be obtained for a basic language are almost always useful when the language is extended with additional constructs. Most of the time, these basic languages are concrete. In this survey we will see many examples where a result for a basic language is very useful for an extended version of the language.

Another reason to focus on concrete process algebra is that it is indeed purely algebraically a neat theory. On the other hand, the theory with a form of abstraction and thus with some special constant such as Milner's silent action ( $\tau$ ) or the empty process  $\varepsilon$  of [Koymans and Vrancken, 1985] is not (yet) stabilized. That is, there are many variants of the theory and it is not clear if there exists a superior variant. For instance, there are two closely related competitive equivalences for the theory with so-called  $\tau$  abstraction: observational congruence [Milner, 1980] and branching bisimulation equivalence [Glabbeek and Weijland, 1989].

To obtain a uniform notation, since the majority of the available concrete process algebras are ACP-like ones, and since the ACP approach is the most algebraical approach, we survey the algebraical part in the ACP-style process algebra of [Bergstra and Klop, 1984a; Bergstra and Klop, 1995]. As for the semantics of the various languages we deviate from the approach of [Bergstra and Klop, 1984a; Bergstra and Klop, 1995] since nowadays many process algebras have an operational semantics in the style of [Plotkin, 1981]. So, we equip all the languages with such an operational semantics. In the articles [Bergstra and Klop, 1982], [Bergstra and Klop, 1984b], BPA, PA, and ACP were introduced with a semantics in terms of projective limit models. When we restrict ourselves to guarded recursion, projective limit models identify exactly the (strongly) bisimilar processes. The projective limit models are an algebraic reformulation of the topological structures used in [Bakker and Zucker, 1982]. Regarding syntax as well as semantics, [Bergstra and Klop, 1982] reformulate [Bakker and Zucker, 1982] in order to allow more efficient algebraic reasoning.

For those readers who want to know more about possibly other approaches to process algebra (with abstraction), we refer to the following four textbooks in the area [Baeten and Weijland, 1990], [Hennessy, 1988], [Hoare, 1985], and [Milner, 1989]; see also section 4.

Finally, we briefly review what can be expected in this survey. The survey is organized into three sections (not counting this section).

The first section (2) describes concrete sequential processes; that is, in this section we even refrain from discussing parallelism. In this section, we will meet and tackle many problems that accompany the design of any algebraic language. Since the languages are simple, it is relatively easy to explain the solutions. The solutions that we obtain for the concrete



sequential processes turn out to be useful for other languages, too. In particular, we will use these solutions in section 3 where we discuss concrete concurrent processes. Lastly, section 4 gives directions for further reading.

*Acknowledgements* We thank the proof readers, Jan Bergstra (University of Amsterdam and Utrecht University) and Jan Willem Klop (CWI and Free University Amsterdam) for their useful comments. Also comments by Twan Basten (Eindhoven University of Technology), Kees Middelburg (PTT Research and Utrecht University), Alban Ponse (University of Amsterdam), and Michel Reniers (Eindhoven University of Technology) were appreciated. Special thanks go to Joris Hillebrand (University of Amsterdam) for his essential help in the final stages of the preparation of the document.

## 2 Concrete sequential processes

In this section we will introduce some basic concepts that can be found in process algebra. We will do this in a modular way. That is, first we treat a basic theory that is the kernel for all the other theories that we will discuss subsequently. The basic theory describes finite, concrete, sequential non-deterministic processes. Then we add features to this kernel that are known to be important in process algebra: for instance, deadlock or recursion to mention a few. Such features make the kernel more powerful for both theoretical and practical purposes. We also show that each feature is a so-called conservative extension of the original theory; thus, we may argue that our approach is modular.

### 2.1 Introduction

In this subsection we give the reader an idea of what can be expected in the subsections of the sequential part of this survey.

We start with the basic language. Once we have treated this language, we will extend it in the following subsections with important features. We discuss the notions of recursion in subsection 2.3, projection in 2.4, deadlock (or inaction) in 2.5, empty process in 2.6, and we discuss the following operators: renaming operators in subsection 2.7, state operators in 2.8 and 2.9, the priority operator in 2.10, and Kleene's binary star operator in 2.11. Next, we focus in subsection 2.12 on an extension with time. Then subsection 2.13 follows with pointers to extensions that we do not discuss in this survey. Finally, we discuss decidability and expressiveness issues in subsection 2.14 for some of the languages introduced.

### 2.2 Basic process algebra

First, we list some preliminaries. Then we treat the basic language of this chapter. Next, we devote subsection 2.2.2 to term rewriting analysis; we discuss a powerful method that we will frequently need subsequently. In

the next, and last, subsection 2.2.3 we discuss an operational semantics for our basic language. In 2.2.3 we also treat a meta-theorem on operational semantics that we will often use in the rest of this survey.

We assume that we have an infinite set  $V$  of variables with typical elements  $x, y, z, \dots$ . A (single sorted) signature  $\Sigma$  is a set of function symbols together with their arity. If the arity of a function symbol  $f \in \Sigma$  is zero we say that  $f$  is a constant symbol. The notion of a term (over  $\Sigma$ ) is defined as expected:  $x \in V$  is a term; if  $t_1, \dots, t_n$  are terms and if  $f \in \Sigma$  is  $n$ -ary then  $f(t_1, \dots, t_n)$  is a term. A term is also called an open term; if it contains no variables we call it closed. We denote the set of closed terms by  $C(\Sigma)$  and the set of open terms by  $O(\Sigma)$  (note that a closed term is also open). We also want to speak about the variables occurring in terms: let  $t \in O(\Sigma)$ ; then  $\text{var}(t) \subseteq V$  is the set of variables occurring in  $t$ .

A substitution  $\sigma$  is a map from the set of variables into the set of terms over a given signature. This map can easily be extended to the set of all terms by substituting for each variable occurring in an open term its  $\sigma$ -image.

### 2.2.1 The theory Basic Process Algebra

We will give the theory Basic Process Algebra or BPA in terms of an equational specification. BPA is due to [Bergstra and Klop, 1982].

**Definition 2.2.1.** An *equational specification*  $(\Sigma, E)$  consists of a set  $\Sigma$  that is a signature and a set of equations of the form  $t_1 = t_2$  where  $t_1$  and  $t_2$  are (open) terms. The equations in  $E$  are often referred to as *axioms*.

Now we give the theory  $\text{BPA} = (\Sigma_{\text{BPA}}, E_{\text{BPA}})$ .

We begin with the signature  $\Sigma_{\text{BPA}}$ . There are two binary operators present in  $\Sigma_{\text{BPA}}$ ; they are denoted  $+$  and  $\cdot$ . The signature  $\Sigma_{\text{BPA}}$  also contains a number of constants with typical names  $a, b, c, \dots$ . We will use the capital letter  $A$  for the set of constants. The set  $A$  can be seen as a parameter of the theory BPA: for each application the set  $A$  will be specified. For now it is only important that there are constants. This ends our discussion of the signature.

The set of equations  $E_{\text{BPA}}$  consists of the five equations A1–5 in table 1. The variables  $x, y$ , and  $z$  in table 1 are universally quantified. They stand for elements of some arbitrary model of BPA. These elements are often called *processes*.

**Remark 2.2.2.** Terms will be denoted according to the same conventions as the usual ones for summation and multiplication. We will often omit the centered dot in a product. The centered dot binds stronger than the plus. Thus,  $xy + z$  means  $(x \cdot y) + z$  and the brackets in  $x(y + z)$  cannot be omitted.

Table 1. BPA.

$x + y = y + x$	A1
$(x + y) + z = x + (y + z)$	A2
$x + x = x$	A3
$(x + y)z = xz + yz$	A4
$(xy)z = x(yz)$	A5

*Intuition* We will give an intuitive meaning of the signature and the axioms respectively. Formal semantics can be found in 2.2.3.

The constants  $a, b, c, \dots$  are called *atomic actions* or *steps*. We consider them as processes, which are not subject to any investigation whatsoever.

We think of the centered dot  $(\cdot)$  as *sequential composition*. The process  $xy$  is the process that first executes the process  $x$  and when (and if) it is completed  $y$  starts.

The sum or *alternative composition*  $x + y$  of two processes  $x$  and  $y$  represents the process that executes either  $x$  or  $y$  but not both.

Now we will discuss the axioms of table 1.

Axiom A1 expresses the *commutativity* of the alternative composition. It says that the choice between  $x$  and  $y$  is the same as the choice between  $y$  and  $x$ .

Axiom A2 expresses the *associativity* of the plus. It says that first choosing between  $x + y$  and  $z$  and then (possibly) a choice between  $x$  and  $y$  is the same as choosing between  $x$  and  $y + z$  and then (possibly) a choice between  $y$  and  $z$ .

Axiom A3 expresses the *idempotency* of the alternative composition. A choice between  $x$  and  $x$  is the same as a choice for  $x$ .

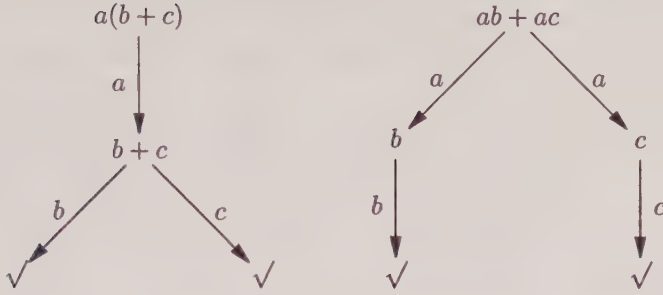
Axiom A4 expresses the *right distributivity* of the sequential composition over the alternative composition. A choice between  $x$  and  $y$  followed by  $z$  is the same as a choice between  $x$  followed by  $z$  and  $y$  followed by  $z$ .

Axiom A5 expresses the associativity of the sequential composition. First  $xy$  followed by  $z$  is the same as first  $x$  followed by  $yz$ .

*Full distributivity* We will explain why only right distributivity is presented in table 1. An axiom that does not appear in BPA is the axiom that expresses the *left distributivity* (LD) of the sequential composition over the alternative composition:

$$\text{LD} \quad x(y + z) = xy + xz.$$

Axioms A4 and LD together would give full distributivity. Axiom LD is not included on intuitive grounds. In the left-hand side of LD the moment of choice is later than in the right-hand side. For in  $a(b + c)$  the choice



**Fig. 1.** Two deduction graphs with the same execution paths  $ab$  and  $ac$  but with different choice moments.

between  $b$  and  $c$  is made after the execution of  $a$ , whereas in  $ab+ac$  first the choice between  $ab$  and  $ac$  must be made and then the chosen term can be executed, as in figure 1, where we depict two deduction graphs. See definition 2.2.23 later on for a formal definition of a deduction graph.

The right-hand side of LD is often called a *non-deterministic choice*, which is a subject of research on its own.

*Structural induction* Structural induction is a basic proof technique in process algebra when closed terms are involved. We will inductively define the class of basic terms. It will turn out that every closed term can be written as a basic term.

**Definition 2.2.3.** An atomic action is a *basic* term. If  $t$  is a basic term and  $a \in A$ , then  $a \cdot t$  is a basic term. If  $t$  and  $s$  are basic terms, then  $t+s$  is a basic term.

**Remark 2.2.4.** If we consider terms to be identical that only differ in the order of the summands, we can see that basic terms have the following form

$$\sum_{i=1}^n a_i \cdot t_i + \sum_{j=1}^m b_j,$$

where  $a_i, b_j \in A$ ,  $1 \leq i \leq n$ ,  $1 \leq j \leq m$ ,  $n+m \geq 1$ , and the  $t_i$  again basic for  $1 \leq i \leq n$ . Here,

$$\sum_{i=1}^k p_i$$

is an abbreviation of  $p_1 + \dots + p_k$ , and if  $n=0, m \geq 1$  we have a term of the form  $b_1 + \dots + b_m$ . Similarly if  $m=0, n \geq 1$ .

In the next proposition we see that if we want to prove a statement correct for all closed terms (see subsection 2.2 for the definition of a closed



term), it suffices to prove it for basic terms. Since they are inductively defined we can use structural induction.

**Proposition 2.2.5.** *Let  $t$  be a closed BPA term. Then there is a basic term  $s$  such that  $\text{BPA} \vdash t = s$ .*

**Proof.** We will use term rewriting analysis to prove 2.2.5. In the next subsection (2.2.2) we will give a short introduction to this theory. We will use the proof of the fact that the term rewriting system of table 2 is strongly normalizing as a running example. Once we know that the term rewriting system of table 2 has this property, it is not difficult to see that given a closed BPA term  $t$ , there is a normal form  $s$ , which is a basic term, and that  $\text{BPA} \vdash t = s$ , which proves the proposition. ■

### 2.2.2 Term rewriting systems

In this subsection we will introduce a result from the field of term rewriting systems that is a powerful tool in process algebra. We will do this by means of an example: we will prove the essential step of proposition 2.2.5 using the result. General references to this theory are [Dershowitz and Jouannaud, 1990] and [Klop, 1992].

**Definition 2.2.6.** A term rewriting system or term reduction system is a pair  $(\Sigma, R)$  with  $\Sigma$  a signature and  $R$  a set of rewriting (or reduction) rules. The reduction rules are of the form  $s \rightarrow t$ , where  $s$  and  $t$  are terms over the signature  $\Sigma$  (we denote this set by  $O(\Sigma)$  with  $O$  for open terms over  $\Sigma$ ). For the terms  $s$  and  $t$  we have two constraints:

- $s$  is not a variable.
- all variables that occur in  $t$  must also occur in  $s$ .

Often, we give reduction rules a name, as in our example below. The one-step reduction relation on terms, also denoted  $\rightarrow$ , is the smallest relation on terms containing the rules that is closed under substitutions and contexts. We denote the transitive-reflexive closure of the one-step reduction relation  $\rightarrow$  by  $\rightarrow^*$ .

**Example 2.2.7.** We give an example of a term reduction system. Let  $T$  be the term reduction system with as signature that of BPA and as a set of reduction rules those in table 2. Note that we do not have rules corresponding to axioms A1 or A2, as these axioms have no clear direction.

A useful property for a term reduction system is that there are no infinite reductions possible. Below we define some more notions.

**Definition 2.2.8.** Let  $(\Sigma, R)$  be a term reduction system and let  $s$  be a  $\Sigma$  term. We say that  $s$  is a normal form if there is no term  $t$  such that  $s \rightarrow t$ . A term  $s$  has a normal form if there exists a normal form  $t$  with  $s \rightarrow^* t$ .

A term  $s_0$  is called strongly normalizing or terminating (SN) if there



exists no infinite series of reductions beginning in  $s_0$ :

$$s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \dots$$

A term reduction system is called strongly normalizing if every term of it is SN.

**Example 2.2.9.** For our running example we have that the process  $a$  is a normal form, the process  $(ab)c$  is not a normal form but has one,  $a(bc)$  (use RA5), and there are no infinite reductions possible. To prove the last statement, we introduce the method of the recursive path ordering following [Klop, 1992].

**Definition 2.2.10.** Let  $(\Sigma, R)$  be a term reduction system. We define  $O^*(\Sigma)$  to be the superset of  $O(\Sigma)$  where some function (and constant) symbols may be marked with an asterisk (\*).

**Example 2.2.11.** A typical element of the superset  $O^*$  of our running example is

$$a \cdot^* (b \cdot c^*).$$

**Definition 2.2.12.** Let  $\Sigma$  be a signature and let  $>$  be a well-founded partial ordering on  $\Sigma$ . Let  $\rightarrow$  be the reduction relation that is defined in the clauses RPO1–5 in table 3.

Let  $s, t \in O^*(\Sigma)$ . We write  $s \cong t$  if  $s$  can be obtained from  $t$  by permuting the arguments of  $t$ .

Let  $s, t \in O^*(\Sigma)$ . We write  $s >_{rpo} t$  if there exists a  $u \in O^*(\Sigma)$  such that  $u \cong t$  and  $s \rightarrow^+ u$ . With  $\rightarrow^+$  we mean the transitive closure of  $\rightarrow$ .

**Example 2.2.13.** Suppose that we have the following ordering on the signature of our running example:  $\cdot > +$ . With this choice of  $>$  we can execute the following reduction:

$$\begin{aligned} (x + y) \cdot z &>_{rpo} (x + y) \cdot^* z \\ &>_{rpo} (x + y) \cdot^* z + (x + y) \cdot^* z \\ &>_{rpo} (x +^* y) \cdot z + (x +^* y) \cdot z \\ &>_{rpo} x \cdot z + y \cdot z. \end{aligned}$$

In the following theorem (due to [Dershowitz, 1987]) we will see that if we

**Table 2.** A term reduction system for BPA.

$x + x \rightarrow x$	RA3
$(x + y)z \rightarrow xz + yz$	RA4
$(xy)z \rightarrow x(yz)$	RA5

have such a reduction for each rewrite rule we have a strongly normalizing term rewriting system.

**Theorem 2.2.14.** *Let  $(\Sigma, R)$  be a term rewriting system with finitely many rewriting rules and let  $>$  be a well-founded ordering on  $\Sigma$ . If  $s >_{rpo} t$  for each rewriting rule  $s \rightarrow t \in R$ , then the term rewriting system  $(\Sigma, R)$  is strongly normalizing.*

The method of the recursive path ordering is not convenient for rewriting rules such as  $(x \cdot y) \cdot z \rightarrow x \cdot (y \cdot z)$ . We will discuss a variant of the above method which is known as the lexicographical variant of the recursive path ordering. The idea is that we give certain function symbols the so-called lexicographical status (the remaining function symbols have the multiset status). The function symbols with a lexicographical status have, in fact, an arbitrary but fixed argument for which this status holds. For instance, we give the sequential composition the lexicographical status for the first argument.

We also have an extra rule to cope with function symbols with a lexicographical status. For a  $k$ -ary function symbol  $H$  with the lexicographical status for the  $i$ th argument we have the following extra rule in table 4. The idea behind this rule is that if the complexity of a dedicated argument is reduced and the complexity of the other arguments increases (but not unboundedly) the resulting term will be seen as less complex as a whole.

**Definition 2.2.15.** Let  $s, t \in O^*(\Sigma)$ . We write  $s >_{lpo} t$  if  $s \rightarrow^+ t$  with  $\rightarrow^+$  this time the transitive closure of the reduction relation defined by the rules RPO1–5 and LPO.

**Example 2.2.16.** Suppose that we have the following ordering on the

**Table 3.** The recursive path ordering.

---

RPO1. Mark head symbol ( $k \geq 0$ )
$H(t_1, \dots, t_k) \rightarrow H^*(t_1, \dots, t_k)$
RPO2. Make copies under smaller head symbol ( $H > G$ , $k \geq 0$ )
$H^*(t_1, \dots, t_k) \rightarrow G(H^*(t_1, \dots, t_k), \dots, H^*(t_1, \dots, t_k))$
RPO3. Select argument ( $k \geq 1$ , $1 \leq i \leq k$ )
$H^*(t_1, \dots, t_k) \rightarrow t_i$
RPO4. Push $*$ down ( $k \geq 1$ , $l \geq 0$ )
$H^*(t_1, \dots, G(s_1, \dots, s_l), \dots, t_k) \rightarrow H(t_1, \dots, G^*(s_1, \dots, s_l), \dots, t_k)$
RPO5. Handling contexts
$s \rightarrow t \implies H(\dots, s, \dots) \rightarrow H(\dots, t, \dots)$

---

signature of our running example  $\cdot > +$ . We give the symbol  $\cdot$  the lexicographical status for the first argument. Consider the following reduction:

$$\begin{aligned}
 (x \cdot y) \cdot z &>_{lpo} (x \cdot y) \cdot^* z \\
 &>_{lpo} (x \cdot^* y) \cdot ((x \cdot y) \cdot^* z) \\
 &>_{lpo} x \cdot ((x \cdot^* y) \cdot z) \\
 &>_{lpo} x \cdot (y \cdot z).
 \end{aligned}$$

Note that we did not use permutation of arguments in the deduction of example 2.2.13. This means that we also have

$$(x + y) \cdot z >_{lpo} x \cdot z + y \cdot z.$$

In the following theorem (due to [Kamin and Lévy, 1980]) we will see that if we have such a reduction for each rewrite rule we also have a strongly normalizing term rewriting system.

**Theorem 2.2.17.** *Let  $(\Sigma, R)$  be a term rewriting system with finitely many rewriting rules and let  $>$  be a well-founded ordering on  $\Sigma$ . If  $s >_{lpo} t$  for each rewriting rule  $s \rightarrow t \in R$ , then the term rewriting system  $(\Sigma, R)$  is strongly normalizing.*

So with the aid of the above theorem we conclude that the term rewriting system of table 2 is strongly normalizing (we leave the case RA3 to the reader). To prove strong normalization we will henceforth confine ourselves to giving a partial ordering  $>$  on the signature and to saying which function symbols do have the lexicographical status (and for which argument).

### 2.2.3 Semantics of basic process algebra

In this subsection we will give an operational semantics of BPA in the style of Plotkin; see [Plotkin, 1981]. The usual procedure to give an operational semantics is to give a table only with so-called transition rules; see, for instance, table 5. In this subsection we will make a small excursion to the so-called general theory of structured operational semantics because in that framework we can formulate general theorems that hold for large classes of languages. The reason for this detour is that we will use such general

**Table 4.** The extra rule for the lexicographical variant of the recursive path ordering. We have that  $H$  has the lexicographical status for the  $i$ th argument.

LPO Reduce $i$ th argument ( $k \geq 1$ , $1 \leq i \leq k$ , $l \geq 0$ )
Let $t \equiv H^*((t_1, \dots, t_{i-1}, G(s_1, \dots, s_l), t_{i+1}, \dots, t_k)$
Then $t \rightarrow H(t, \dots, t, G^*(s_1, \dots, s_l), t, \dots, t)$

results many times in this chapter.

To start with, we define the notion of a term deduction system, which is taken from [Baeten and Verhoef, 1993]. It is a modest generalization of the notion of a transition system specification that originates from [Groote and Vaandrager, 1992]. The idea of a term deduction system is that it can be used not only to define a transition relation but also to define unary predicates on states that turn out to be useful. See table 5 for a typical term deduction system; it is a definition of both transition relations  $\xrightarrow{a}$  and unary predicates  $\xrightarrow{a}\checkmark$  for each  $a \in A$ .

**Definition 2.2.18.** A term deduction system is a structure  $(\Sigma, D)$  with  $\Sigma$  a signature and  $D$  a set of deduction rules. The set  $D = D(T_p, T_r)$  is parameterized with two sets, which are called respectively the set of predicate symbols and the set of relation symbols. Let  $P \in T_p$  and  $R \in T_r$ , and  $s, t, u \in O(\Sigma)$ . We call expressions  $Ps$  and  $tRu$  formulas. A deduction rule  $d \in D$  has the form

$$\frac{H}{C}$$

with  $H$  a set of formulas and  $C$  a formula. We call the elements of  $H$  the hypotheses of  $d$  and we call the formula  $C$  the conclusion of  $d$ . If the set of hypotheses of a deduction rule is empty we call such a rule an axiom. We denote an axiom simply by its conclusion provided that no confusion can arise. Notions such as “substitution”, “*var*”, or “closed” extend to formulas and deduction rules as expected.

**Example 2.2.19.** Let  $T(\text{BPA})$  be the term deduction system consisting of the signature of BPA and the deduction rules of table 5. As relation symbols we have the transition relations and as predicate symbols we have the successful termination predicates. The intuitive idea of  $s \xrightarrow{a} s'$  is that, for example, a machine in state  $s$  can evolve into state  $s'$  by executing step  $a$ . The intended meaning of  $s \xrightarrow{a}\checkmark$  is that this machine in state  $s$  can terminate successfully by executing  $a$ ; the symbol  $\checkmark$  (pronounced tick) stands for successful termination.

Next, we give the definition of a proof of a formula from a set of deduction rules. This definition is taken from [Groote and Vaandrager, 1992].

**Definition 2.2.20.** Let  $T = (\Sigma, D)$  be a term deduction system. A proof of a formula  $\psi$  from  $T$  is a well-founded upwardly branching tree of which the nodes are labelled by formulas such that the root is labelled with  $\psi$  and if  $\chi$  is the label of a node  $q$  and  $\{\chi_i : i \in I\}$  is the set of labels belonging to the nodes  $\{q_i : i \in I\}$  directly above  $q$  ( $I$  some is index set) then there is a deduction rule

$$\frac{\{\phi_i : i \in I\}}{\phi}$$

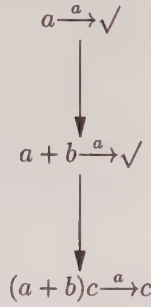


Fig. 2. A proof

and a substitution  $\sigma : V \longrightarrow O(\Sigma)$  such that  $\sigma(\phi) = \chi$  and  $\sigma(\phi_i) = \chi_i$  for  $i \in I$ .

If a proof of  $\psi$  exists, we say that  $\psi$  is provable from  $T$ , notation  $T \vdash \psi$ .

**Example 2.2.21.** It is not difficult to verify that the tree in figure 2 is a proof of the transition  $(a + b)c \xrightarrow{a} c$ .

Next, we define the notion of a deduction graph. It generalizes the well-known notion of a labelled state transition diagram in the sense that it can also handle unary predicates on states. First, we need a reachability definition.

**Definition 2.2.22.** Let  $T$  be a term deduction system and let  $s$  and  $t$  be terms. We define a binary relation  $\rho$  as the transitive reflexive closure of the binary relation  $\{(s, t) \mid \exists R : T \vdash sRt\}$ . If  $s\rho t$  we say that from  $s$  we can reach  $t$ , or that  $t$  is reachable from  $s$ .

**Definition 2.2.23.** Let  $T$  be a term deduction system. The deduction graph of a closed term  $s$  is a labelled graph that is obtained as follows. The nodes of this graph are the terms that can be reached from  $s$ ; the

Table 5. Derivation rules of  $T(\text{BPA})$ .

$a \xrightarrow{a} \surd$	
$\frac{x \xrightarrow{a} x'}{x + y \xrightarrow{a} x'}$	$\frac{y \xrightarrow{a} y'}{x + y \xrightarrow{a} y'}$
$\frac{x \xrightarrow{a} \surd}{x + y \xrightarrow{a} \surd}$	$\frac{y \xrightarrow{a} \surd}{x + y \xrightarrow{a} \surd}$
$\frac{x \xrightarrow{a} x'}{xy \xrightarrow{a} x'y}$	$\frac{x \xrightarrow{a} \surd}{xy \xrightarrow{a} y}$



labels of nodes are sets of predicate symbols. The edges of a deduction graph are labelled by relation symbols. Let  $t$  be a node of the deduction graph of  $s$ ; then there is a label  $\{P : T \vdash Pt\}$  attached to this node. Let  $t$  and  $t'$  be nodes of the deduction graph of  $s$ . Then there exists an edge labelled by  $R$  from  $t$  to  $t'$  in the deduction graph of  $s$  if and only if we have  $T \vdash Rt'$ . See figure 3 for examples. Observe that we are a bit sloppy there: we identify the edges of the graph with its labels; that is, we render an edge

$$\xrightarrow{\quad} \rightarrow$$

simply as  $\xrightarrow{a}$ . Moreover, we depict a predicate  $\xrightarrow{b}\surd$  as a  $b$ -labelled edge to a node  $\surd$ .

Next, we define the notion of a structured state system [Baeten and Verhoef, 1993]. It is a generalization of the well-known notion of a labelled transition system.

**Definition 2.2.24.** A structured state system is a triple  $(S, S_p, S_r)$  where  $S$  is a set of states,  $S_p$  is a subset of the power set of  $S$ , and  $S_r$  is a subset of the power set of  $S \times S$ . The sets  $S_p$  and  $S_r$  are called respectively the set of predicates and the set of relations.

A term deduction system induces in a natural way a structured state system.

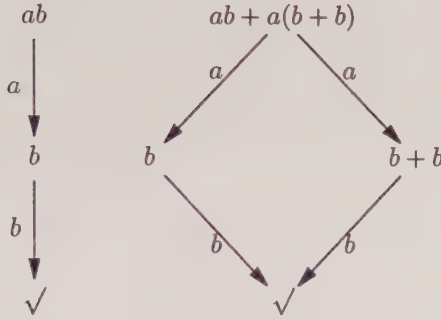
**Definition 2.2.25.** Let  $T = (\Sigma, D)$  be a term deduction system and let  $D = D(T_p, T_r)$ . The structured state system  $G$  induced by  $T$  has as its set of states  $S = C(\Sigma)$ ; the predicates and relations are the following:

$$\begin{aligned} S_p &= \left\{ \{t \in C(\Sigma) \mid T \vdash Pt\} \mid P \in T_p \right\}, \\ S_r &= \left\{ \{(s, t) \in C(\Sigma) \times C(\Sigma) \mid T \vdash sRt\} \mid R \in T_r \right\}. \end{aligned}$$

**Example 2.2.26.** Let  $L = L(\text{BPA})$  be the structured state system induced by the term deduction system  $T(\text{BPA})$  from example 2.2.19. In figure 3 we depict two deduction graphs of the terms  $ab + a(b + b)$  and  $ab$ .

Both terms represent the same behaviour: first  $a$  is executed, then  $b$ , and then both systems terminate successfully. However, they do not have the same deduction graphs as we can see in figure 3. So the set of deduction graphs is not directly a model of BPA since in that system we want the alternative composition to be idempotent.

Many different equivalence notions have been defined in order to identify states that have the same behaviour; see [Glabbeek, 1990] and [Glabbeek, 1993] for a systematic approach. The finest among them is the so-called strong bisimulation equivalence of [Park, 1981]. We will take the formulation of [Baeten and Verhoef, 1993] for this well-known notion.



**Fig. 3.** Two deduction graphs.

It will turn out that the two deduction graphs of example 2.2.26 are bisimilar.

**Definition 2.2.27.** Let  $G = (S, S_p, S_r)$  be a structured state system. A relation  $B \subseteq S \times S$  is called a (strong) bisimulation if for all  $s, t \in S$  with  $sBt$  the following conditions hold. For all  $R \in S_r$

$$\begin{aligned} \forall s' \in S (sRs' \Rightarrow \exists t' \in S : tRt' \wedge s'Bt'), \\ \forall t' \in S (tRt' \Rightarrow \exists s' \in S : sRs' \wedge s'Bt'), \end{aligned}$$

and for all  $P \in S_p$

$$Ps \Leftrightarrow Pt.$$

The first two conditions are known as the transfer property. Two states  $s$  and  $t$  in  $S$  are bisimilar in the structured state system  $G$  if there exists a bisimulation relation containing the pair  $(s, t)$ . The notation is  $s \sim_G t$  or  $s \sim t$  provided that no confusion can arise.

Note that bisimilarity is an equivalence relation, called a bisimulation equivalence.

**Example 2.2.28.** Let  $L = L(\text{BPA})$  be the structured state system of example 2.2.26. It is not hard to see that the two states  $ab + a(b + b)$  and  $ab$  are bisimilar. We graphically depict the bisimulation relation by connecting its pairs with a dashed line as in figure 4.

When two states in a deduction graph are bisimilar, we also call the corresponding terms bisimilar.

An example of two deduction graphs that are *not* bisimilar can be found in figure 1.

Considered as deduction graphs, the two processes  $x = ab$  and  $y = ab + a(b + b)$  are not equal, but from a process algebraic point of view, we want them to be. That is, they both first execute the atomic process  $a$  and then

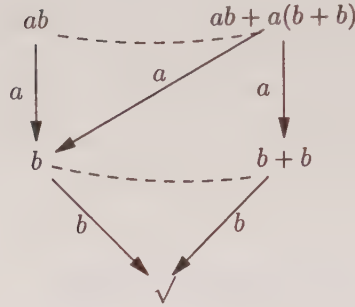


Fig. 4. Bisimilar deduction graphs.

the  $b$ . So we would like to have a model for which  $ab = ab + a(b + b)$ . The usual approach to obtain this is to work with an equivalence relation and to identify equivalent objects. We then say that the objects are equal modulo this equivalence relation. If  $x$  is a process and  $\sim$  denotes bisimulation equivalence, the equivalence class is defined as  $[x] = \{y : x \sim y\}$ . So, in the above example the two processes  $x = ab$  and  $y = ab + a(b + b)$  are equal modulo bisimulation equivalence:  $[x] = [y]$ . Now it would be very nice if the equivalence class is independent of the chosen representative. If this is the case, we can easily define our process algebra operators on these classes. For instance, the alternative composition can be defined as  $[x] + [y] = [x + y]$ . In general, the assumption that a relation is an equivalence relation is too weak for this purpose. The additional property that does the job is called congruency. In the next definition we define this well-known notion.

**Definition 2.2.29.** Let  $\Sigma$  be a signature. An equivalence relation  $R$  on the set of closed  $\Sigma$  terms is called a congruence if for all  $n$ -ary function symbols  $f \in \Sigma$  we have

$$x_1 R y_1, \dots, x_n R y_n \implies f(x_1, \dots, x_n) R f(y_1, \dots, y_n)$$

where  $x_1, y_1, \dots, x_n, y_n$  are closed  $\Sigma$  terms.

Next, we define some syntactical constraints on the rules of a term deduction system for which it can be proved that if a term deduction system satisfies these constraints then strong bisimulation equivalence will always be a congruence. Below we discuss the so-called *path* format; this stands for “predicates and *tyft/tyxt* hybrid format” and was proposed by [Baeten and Verhoef, 1993]. It is a modest generalization of the *tyft/tyxt* format originating from [Groote and Vaandrager, 1992]. The name *tyft/tyxt* refers to the syntactical form of the deduction rules.

We refer to [De Simone, 1985] for the first paper that discusses syntactical constraints on operational rules. Nowadays, the syntactical constraints

formulated in that paper are often referred to as the “De Simone format”.

**Definition 2.2.30.** Let  $T = (\Sigma, D)$  be a term deduction system with  $D = D(T_p, T_r)$ . Let  $I$  and  $J$  in the following be index sets of arbitrary cardinality, let  $t_i, s_j, t \in O(\Sigma)$  for all  $i \in I$  and  $j \in J$ , let  $P_j, P \in T_p$  be predicate symbols for all  $j \in J$ , and let  $R_i, R \in T_r$  be relation symbols for all  $i \in I$ . A deduction rule  $d \in D$  is in *path* format if it has one of the following four forms:

$$\frac{\{P_j s_j : j \in J\} \cup \{t_i R_i y_i : i \in I\}}{f(x_1, \dots, x_n) R t}$$

with  $f \in \Sigma$  an  $n$ -ary function symbol,  $X = \{x_1, \dots, x_n\}$ ,  $Y = \{y_i : i \in I\}$ , and  $X \cup Y \subseteq V$  a set of distinct variables;

$$\frac{\{P_j s_j : j \in J\} \cup \{t_i R_i y_i : i \in I\}}{x R t}$$

with  $X = \{x\}$ ,  $Y = \{y_i : i \in I\}$ , and  $X \cup Y \subseteq V$  a set of distinct variables;

$$\frac{\{P_j s_j : j \in J\} \cup \{t_i R_i y_i : i \in I\}}{P f(x_1, \dots, x_n)}$$

with  $f \in \Sigma$  an  $n$ -ary function symbol,  $X = \{x_1, \dots, x_n\}$ ,  $Y = \{y_i : i \in I\}$ , and  $X \cup Y \subseteq V$  a set of distinct variables or

$$\frac{\{P_j s_j : j \in J\} \cup \{t_i R_i y_i : i \in I\}}{P x}$$

with  $X = \{x\}$ ,  $Y = \{y_i : i \in I\}$ , and  $X \cup Y \subseteq V$  a set of distinct variables.

If in the above four cases  $\text{var}(d) = X \cup Y$  we say that the deduction rule  $d$  is pure.

We say that a term deduction system is in *path* format if all its deduction rules are in *path* format. We say that a term deduction system is pure if all its rules are pure.

Next, we formulate the congruence theorem for the *path* format. It is taken from [Baeten and Verhoef, 1993]. There the so-called well-founded subcase is proved. [Fokkink, 1994] showed that this requirement is not necessary, thus yielding the following result. Note that we do not use the notion pure in the theorem below. We just define it since we will need this notion later on when we will have our second excursion into the area of general SOS theory.

**Theorem 2.2.31.** Let  $T = (\Sigma, D)$  be a term deduction system in *path* format. Then strong bisimulation equivalence is a congruence for all function symbols occurring in  $\Sigma$ .

**Lemma 2.2.32.** *Let  $T(\text{BPA})$  be the term deduction system that we defined in example 2.2.19. Then bisimulation equivalence is a congruence on the set of closed BPA terms.*

**Proof.** It is easy to see that the operational semantics given in table 5 is in *path* format, so with theorem 2.2.31 we immediately find the proof of this lemma. ■

It follows from lemma 2.2.32 that we can take the quotient of the algebra of closed BPA terms with respect to bisimulation equivalence and that the operators of BPA can be defined on this quotient by taking representatives. Next, we will show that this quotient algebra is a model of BPA. We call this property soundness; that is, if two closed terms  $x$  and  $y$  are provably equal,  $\text{BPA} \vdash x = y$ , then we also have that  $x$  and  $y$  are bisimilar,  $x \sim y$ .

**Theorem 2.2.33.** *The set of closed BPA terms modulo bisimulation equivalence is a model of BPA.*

**Proof.** Since bisimulation equivalence is a congruence, we only need to verify the soundness of each separate axiom. We check A1 (see table 1). Let  $x$  and  $y$  be closed BPA terms. We have to show that  $x + y$  is bisimilar with  $y + x$  (using the rules of table 5). It is not hard to see that the relation that contains the pair  $(x + y, y + x)$  and the diagonal of  $S \times S$  is a bisimulation. The cases A2–4 are treated analogously. It remains to check A5. It is easy to see that the relation containing all the pairs of the form  $((xy)z, x(yz))$  and the diagonal of  $S \times S$  is a bisimulation. ■

Next we will show that BPA is a complete axiomatization of the set of closed terms modulo bisimulation equivalence. We recall that an axiomatization is complete if bisimilar  $x$  and  $y$  are provably equal with these axioms. Note that we only talk about closed process terms here: complete axiomatizations for open terms are much more difficult, see e.g. [Groote, 1990a]. But first we will need some preliminaries to prove this; they are listed in the next lemma.

**Lemma 2.2.34.** *Let  $x$  and  $y$  be closed BPA terms and let  $n(z)$  be the number of symbols of a closed BPA term  $z$ . Then we have:*

- (i)  $T(\text{BPA}) \vdash x \xrightarrow{a} \sqrt{\phantom{x}} \implies \text{BPA} \vdash x = a + x,$
- (ii)  $T(\text{BPA}) \vdash x \xrightarrow{a} y \implies \text{BPA} \vdash x = ay + x,$
- (iii)  $T(\text{BPA}) \vdash x \xrightarrow{a} y \implies n(x) > n(y).$

**Proof.** Easy. Use induction on the depth of the proof. ■

**Theorem 2.2.35.** *The axiom system BPA is a complete axiomatization of the set of closed BPA terms modulo bisimulation equivalence.*

**Proof.** Let  $x$  and  $y$  be bisimilar closed BPA terms, notation  $x \sim y$ . We have to prove that  $\text{BPA} \vdash x = y$ . With the aid of proposition 2.2.5 and



theorem 2.2.33 it is enough to prove this for basic terms. By symmetry, it is even enough to prove that for basic terms  $x$  and  $y$

$$x + y \sim y \implies \text{BPA} \vdash x + y = y.$$

We will prove this with induction on  $n = n(x) + n(y)$ . First, let  $x = a$ . Then  $y \xrightarrow{a} \surd$ , so with lemma 2.2.34 we find that  $x + y = y$ . This proves the basis of our induction. Now suppose that  $x = ax'$ . Then  $x + y \xrightarrow{a} x'$ , so there is a  $y'$  with  $y \xrightarrow{a} y'$  and  $x' \sim y'$ . But then also  $x' + y' \sim y'$  and  $y' + x' \sim x'$  and with induction we find  $x' + y' = y'$  and  $y' + x' = x'$ . So  $x' = y'$ . Now  $x + y = ax' + y = ay' + y = y$  with lemma 2.2.34. Now suppose that  $x = x' + x''$ . Since  $x + y \sim y$ , we also have  $x' + y \sim y$  and  $x'' + y \sim y$ . By induction  $x' + y = y$  and  $x'' + y = y$ . So  $x + y = x' + x'' + y = y + y = y$ . ■

## 2.3 Recursion in BPA

In this subsection we will add recursion to the theory BPA.

**Definition 2.3.1.** Let  $V$  be a set of variables. A *recursive specification*  $E = E(V)$  is a set of equations

$$E = \{X = s_X(V) : X \in V\},$$

where each  $s_X(V)$  is a BPA term that only contains variables of  $V$ . These equations are called *recursion equations*. A recursion equation is called a *recursive equation* if it has the form  $X = s(X)$  where  $s(X)$  only contains the variable  $X$ . By convention, we use capital letters  $X, Y, \dots$  for variables bound in a recursive specification.

**Example 2.3.2.**  $E_1 = \{X = Xa + a\}$  and  $E_2 = \{Y = aY\}$  are examples of recursive specifications.

**Definition 2.3.3.** A *solution* of a recursive equation is a process in some model of BPA such that its substitution in the recursive equation is a true statement in that model.

A *solution*  $\{\langle X \mid E \rangle : X \in V\}$  of a recursive specification  $E(V)$  is a set of processes in some model of BPA such that replacing each variable  $X$  by  $\langle X \mid E \rangle$  in the recursion equations of  $E(V)$  yields true statements in that model. Mostly, we are interested in one particular variable  $X \in V$ . Abusing terminology we call  $\langle X \mid E \rangle$  the solution of  $E$ . Moreover, abusing notation we often write  $X$  for  $\langle X \mid E \rangle$ .

**Remark 2.3.4.** In CCS [Milner, 1980; Milner, 1989] and CSP [Hoare, 1985] the so-called  $\mu$  notation is used: if  $x = t(x)$  is a recursive equation, then  $\mu x.t(x)$  is a process satisfying this equation.

**Definition 2.3.5.** Let  $E = E(V)$  be a recursive specification and let  $t$  be an open BPA term. Then  $\langle t \mid E \rangle$  is the process  $t$  with all variables  $X$  both occurring in  $t$  and  $V$  replaced by the processes  $\langle X \mid E \rangle$ .

At this point we have all the necessary definitions to define the equational specification  $\text{BPAREC}$ , in which  $\text{rec}$  is an abbreviation for recursion. The signature of  $\text{BPAREC}$  consists of the signature of BPA plus for all recursive specifications  $E(V)$  and for all  $X \in V$  a constant  $\langle X \mid E \rangle$ . The axioms of  $\text{BPAREC}$  consist of the axioms of BPA plus for all recursive specifications  $E(V) = \{X = s_X : X \in V\}$  and for all  $X \in V$  an equation  $\langle X \mid E \rangle = \langle s_X \mid E \rangle$ .

**Definition 2.3.6.** Let  $s$  be a term over BPA containing a variable  $X$ . We call an occurrence of  $X$  in  $s$  *guarded* if  $s$  has a subterm of the form  $a \cdot t$  with  $t$  a BPA term containing this occurrence of  $X$ ; in this case we call an atomic action  $a \in A$  a *guard* (of  $X$  in  $s$ ). Otherwise we call the occurrence of  $X$  in  $s$  *unguarded*.

**Definition 2.3.7.** We call a term *completely guarded* if all occurrences of all its variables are guarded.

We call a term *guarded* if we can rewrite it to a completely guarded term by use of the axioms. Otherwise, a term is called *unguarded*.

**Example 2.3.8.** The term  $aX + bX$  is completely guarded,  $(a + b)X$  is guarded but not completely guarded, and  $Xa + Xb$  is unguarded.

**Definition 2.3.9.** We call a recursive specification *completely guarded* if the right-hand sides of its recursion equations are completely guarded.

We call a recursive specification *guarded* if we can rewrite it to a completely guarded recursive specification by use of the axioms and/or its recursion equations. Otherwise, a recursive specification is called *unguarded*.

**Example 2.3.10.** In example 2.3.2, the recursive specification  $E_1$  is unguarded and  $E_2$  is completely guarded.

The next two definitions are taken from [Bergstra and Klop, 1986].

**Definition 2.3.11.** The (*restricted*) *recursive definition principle* is the assumption that every (guarded) recursive specification has a solution. We refer to this assumption as  $\text{RDP}^{(-)}$ .

**Definition 2.3.12.** The *recursive specification principle* (RSP) is the assumption that every guarded recursive specification has at most one solution.

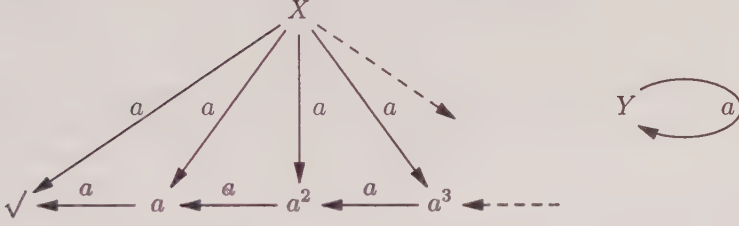
Together,  $\text{RDP}^-$  and RSP say that a guarded recursive specification has a *unique* solution.

*Semantics* The semantics of  $\text{BPAREC}$  can be given completely analogously to the semantics for BPA that we gave in subsection 2.3.

We consider the term deduction system  $T(\text{BPAREC})$  with as signature the signature of  $\text{BPAREC}$  and as rules the rules in table 5 together with those in table 6. Bisimulation equivalence is a congruence on the structured state system  $L(\text{BPAREC})$  induced by  $T(\text{BPAREC})$ ; see 2.2.31. So on the quotient of

**Table 6.** Derivation rules for recursion ( $X = s_X \in E$ ).

$\frac{\langle s_X \mid E \rangle \xrightarrow{a} \surd}{\langle X \mid E \rangle \xrightarrow{a} \surd}$	$\frac{\langle s_X \mid E \rangle \xrightarrow{a} y}{\langle X \mid E \rangle \xrightarrow{a} y}$
-----------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------

**Fig. 5.** Two deduction graphs of the processes  $X$  and  $Y$  that can be found in example 2.3.2.

the algebra of closed BPArec terms with respect to bisimulation equivalence we can define the operators of BPArec by taking representatives. The next theorem states that this quotient algebra is a model of BPArec.

**Theorem 2.3.13.** *The set of closed BPArec terms modulo bisimulation equivalence is a model of BPArec. Moreover, it satisfies RDP and RSP.*

**Proof.** As bisimulation equivalence is a congruence we only need to check the soundness of each axiom. The axioms A1–5 are treated in exactly the same way as in theorem 2.2.33. Equations concerning recursion are treated in the same way as A1.

Let  $E(V)$  be a recursive specification. Then  $\{[\langle X \mid E \rangle] : X \in V\}$  is a solution. See below example 2.2.28 for the  $[\cdot]$  notation.

The proof that this model satisfies RSP will be postponed until we have discussed the combination of recursion and so-called projections. See theorem 2.4.36 for a proof. ■

**Example 2.3.14.** In figure 5, we depict two deduction graphs of the solutions of the two recursive specifications of example 2.3.2. It is not hard to see that  $X \xrightarrow{a} \surd$  and  $X \xrightarrow{a} a^n$  for all  $n \geq 1$ . We can think of the process  $X$  as the infinite sum  $\sum_{n < \omega} a^n$  and we can think of the process  $Y$  as the infinite product  $a^\omega$ . Note that  $X + Y$  and  $X$  are not bisimilar since  $X + Y$  can do infinitely many  $a$  steps, whereas  $X$  can perform only finitely many  $a$  steps.

## 2.4 Projection in BPA

In subsection 2.3 we introduced guarded recursive specifications. They are mainly used to specify infinite processes such as a counter: see exam-

Table 7. Projection.

$\pi_n(a) = a$	PR1
$\pi_1(ax) = a$	PR2
$\pi_{n+1}(ax) = a\pi_n(x)$	PR3
$\pi_n(x + y) = \pi_n(x) + \pi_n(y)$	PR4

ple 2.14.5. With the principles RDP and RSP we can prove statements involving such infinite processes. See example 3.6.1 for an application of RSP. In this subsection we will introduce another method for this purpose. The idea is that we approximate an infinite process by its finite projections. The finite projections for their part turn out to be closed terms and they can therefore be taken care of by structural induction (see also 2.2 for structural induction). This material is based on the paper [Bergstra and Klop, 1982].

We will define the equational specification  $\text{BPA} + \text{PR}$ , in which PR is an abbreviation for projection.

The signature of  $\text{BPA} + \text{PR}$  consists of the signature of BPA plus for each  $n \geq 1$  a unary function  $\pi_n$  that is called a *projection operator of order  $n$*  or the  *$n$ th projection*. The axioms of  $\text{BPA} + \text{PR}$  consist of the axioms of table 7 and the axioms of BPA; we call the axioms PR1–4. In table 7 we have  $n \geq 1$ , the letter  $a$  ranges over all the atomic actions, and the variables  $x$  and  $y$  are universally quantified.

We will now discuss the axioms of table 7.

The idea of projections is that we want to be able to cut off a process at a certain depth. An atomic action is intrinsically the most “shallow” process so we cannot cut off more. Axiom PR1 expresses this: a projection operator is invariant on the set of atomic actions.

The subscript  $n$  of the projection operator serves as a counter for the depth of a process. Axioms PR2 and PR3 illustrate how this counter can be decremented.

Axiom PR4 says that the projection operator distributes over the alternative composition: choosing an alternative does not alter the counter of the projection operator.

*Proof rule* We will discuss a proof rule expressing that a process is determined by its finite projections. This rule is due to [Bergstra and Klop, 1986].

**Definition 2.4.1.** Let  $x$  and  $y$  be processes. The *approximation induction principle* (AIP) is the following assumption. If for all  $n \geq 1$  we have  $\pi_n(x) = \pi_n(y)$  then  $x = y$ .

**Remark 2.4.2.** In the presence of recursion we will define a more restric-

tive version of this principle; see definition 2.4.28.

The following theorem states that projection operators can be eliminated from closed terms. To prove this we will use a method that we briefly explained in subsection 2.2.2. First, we define what we mean by the elimination of operators.

**Definition 2.4.3.** Let  $L = (\Sigma, E)$  and  $L_0 = (\Sigma_0, E_0)$  be two equational specifications with  $\Sigma_0 \subseteq \Sigma$ . If for all  $s \in C(\Sigma)$  there is a  $t \in C(\Sigma_0)$  such that  $L \vdash s = t$  we say that  $L$  has the elimination property (for  $L_0$ ).

**Theorem 2.4.4.** *The equational specification  $\text{BPA} + \text{PR}$  has the elimination property for  $\text{BPA}$ .*

**Proof.** It is not hard to show that the term rewriting system of table 8 is strongly normalizing with the lexicographical variant of the recursive path ordering that we treated in subsection 2.2.2. We confine ourselves to giving a partial ordering  $<$  on the elements of the signature of  $\text{BPA} + \text{PR}$ .

$$+ < \cdot < \pi_1 < \pi_2 < \dots$$

Moreover, we give the sequential composition  $\cdot$  the lexicographical status for the first argument. It is straightforward to show that the left-hand side of each rewrite rule is  $>_{lpo}$  than its right-hand side. Now, apply theorem 2.2.17.

Now it is not hard to see that a normal form (with respect to the term rewriting system in table 8) of a closed  $\text{BPA} + \text{PR}$  term must be a basic  $\text{BPA}$  term, which proves the theorem. ■

Next, we formulate a traditional theorem in process algebra. It states that the term rewriting system associated to the equational specification  $\text{BPA} + \text{PR}$  behaves neatly: it terminates modulo the equations without a clear direction (viz. the commutativity and the associativity of alternative

**Table 8.** A term rewriting system for  $\text{BPA} + \text{PR}$ .

$x + x \rightarrow x$	RA3
$(x + y)z \rightarrow xz + yz$	RA4
$(xy)z \rightarrow x(yz)$	RA5
$\pi_n(a) \rightarrow a$	RPR1
$\pi_1(ax) \rightarrow a$	RPR2
$\pi_{n+1}(ax) \rightarrow a\pi_n(x)$	RPR3
$\pi_n(x + y) \rightarrow \pi_n(x) + \pi_n(y)$	RPR4



composition) and it is confluent modulo these equations. In term rewriting theory, this is expressed by saying that the term rewriting system is complete. Incidentally, note that we proved in theorem 2.4.4 that the associated term rewriting system terminates, but not that it terminates modulo the equations A1 and A2.

The main application of the next result is that it is usually used to prove the conservativity of BPA + PR over BPA (an extension is conservative if no new identities can be derived between original terms in the extended system). The proof of this term rewriting theorem requires much term rewriting theory, which is beyond the scope of this chapter. For more information on these term rewriting techniques we refer to [Jouannaud and Muñoz, 1984] and [Jouannaud and Kirchner, 1986]. Nevertheless, we want to mention the theorem anyway, since it has an importance of its own, for instance for implementational purposes. We will prove the conservativity with an alternative method that we will explain in the next subsection; see subsection 2.4.1.

**Theorem 2.4.5.** *The term rewriting system of table 8 is confluent modulo the equations A1 and A2. Therefore, it has unique normal forms modulo the equations A1 and A2.*

The next theorem states that for a closed term  $s$  the sequence

$$\pi_1(s), \pi_2(s), \dots$$

converges to the term  $s$  itself. It is a nice example of the use of structural induction.

**Proposition 2.4.6.** *Let  $t$  be a closed BPA + PR term. Then there is an  $n \geq 1$  such that for all  $k \geq n$  we have  $\text{BPA} + \text{PR} \vdash \pi_k(t) = t$ .*

**Proof.** It suffices to prove this proposition for basic BPA terms (use theorem 2.4.4). We will use the technique of structural induction that we discussed on page 155. So, we will use the inductive definition of a basic term (2.2.3).

Let  $t$  be an atomic action. Take  $n = 1$  and use axiom PR1.

Let  $t = a \cdot s$  with  $a \in A$  and  $s$ . Suppose that the proposition holds for  $s$ . Then there is an  $n' \geq 1$  with  $\pi_k(s) = s$  for all  $k \geq n'$ . Take  $n = n' + 1$  and use axiom PR3.

Let  $t = s + r$ . Suppose that the proposition holds for  $s$  and  $r$ . Then there are  $n'$  and  $n''$  with the desired properties for  $s$  and  $r$ . Take  $n = \max(n', n'')$  and use axiom PR4. ■

*Semantics* The semantics of BPA + PR can be given in the same way as the semantics for BPA.

We consider the term deduction system  $T(\text{BPA} + \text{PR})$  with as signature the signature of BPA + PR and as deduction rules the rules in tables 5

and 9. Bisimulation equivalence is a congruence on the structured state system induced by  $T(\text{BPA} + \text{PR})$ ; see 2.2.31. So the quotient of closed  $\text{BPA} + \text{PR}$  terms with respect to bisimulation equivalence is well-defined; that is, the operators of  $\text{BPA} + \text{PR}$  can be defined on this quotient by taking representatives. The following theorem states that this quotient is a model of  $\text{BPA} + \text{PR}$ .

**Theorem 2.4.7.** *The set of closed  $\text{BPA} + \text{PR}$  terms modulo bisimulation equivalence is a model of  $\text{BPA} + \text{PR}$  and AIP.*

**Proof.** Strong bisimulation equivalence is a congruence, so to prove the soundness of the axiomatization  $\text{BPA} + \text{PR}$  we just need to check the soundness of the separate axioms. The axioms A1–5 are treated exactly the same as in theorem 2.2.33. So we only need to check the axioms of table 7. The relation between  $\pi_n(a)$  and  $a$  is a bisimulation, so PR1 holds. Axiom PR2 is treated analogously. Axioms PR3 and PR4 are treated as A1.

With proposition 2.4.6 we see that AIP holds. ■

### 2.4.1 Conservativity

Here, we will explain how to prove the conservativity of  $\text{BPA} + \text{PR}$  over  $\text{BPA}$  without using theorem 2.4.5. We recall that the main application of theorem 2.4.5 is that we can prove that adding the projection operators does not yield new identities between  $\text{BPA}$  terms. This important property is called conservativity. We did not give a proof of theorem 2.4.5, but instead we will provide an alternative powerful method for proving the conservativity of  $\text{BPA} + \text{PR}$  over  $\text{BPA}$  (and all the other conservativity theorems in this chapter). This method is based on the format of the operational rules of both systems rather than on a term rewriting analysis. So, we will make a second expedition into the area of general theory on structured operational semantics and we will illustrate the theory with a running example. This example will yield, of course, the conservativity of  $\text{BPA} + \text{PR}$  over  $\text{BPA}$ . We will base ourselves on [Verhoef, 1994b].

First we formalize how we can join two given signatures.

**Definition 2.4.8.** Let  $\Sigma_0$  and  $\Sigma_1$  be two signatures. If for all operators  $f \in \Sigma_0 \cap \Sigma_1$  the arity of  $f$  in  $\Sigma_0$  is the same as its arity in  $\Sigma_1$  then the sum of  $\Sigma_0$  and  $\Sigma_1$ , notation  $\Sigma_0 \oplus \Sigma_1$ , is defined and is equal to the signature  $\Sigma_0 \cup \Sigma_1$ .

**Example 2.4.9.** Let  $\Sigma_0 = A \cup \{+, \cdot\}$  and  $\Sigma_1 = \Sigma_0 \cup \{\pi_n : n \geq 1\}$  be

**Table 9.** Derivation rules for projections.

$\frac{x \xrightarrow{a} \surd}{\pi_n(x) \xrightarrow{a} \surd}$	$\frac{x \xrightarrow{a} x'}{\pi_1(x) \xrightarrow{a} \surd}$	$\frac{x \xrightarrow{a} x'}{\pi_{n+1}(x) \xrightarrow{a} \pi_n(x')}$
------------------------------------------------------------------	---------------------------------------------------------------	-----------------------------------------------------------------------

signatures. Then the sum  $\Sigma_0 \oplus \Sigma_1$  is defined and equals the signature of BPA + PR. Note that these signatures are not disjoint.

Next, we define how to “add” two operational semantics.

**Definition 2.4.10.** Let  $T^i = (\Sigma_i, D_i)$  be term deduction systems with predicate and relation symbols  $T_p^i$  and  $T_r^i$  respectively ( $i = 0, 1$ ). Let  $\Sigma_0 \oplus \Sigma_1$  be defined. The sum of  $T^0$  and  $T^1$ , notation  $T^0 \oplus T^1$ , is the term deduction system  $(\Sigma_0 \oplus \Sigma_1, D_0 \cup D_1)$  with predicate and relation symbols  $T_p^0 \cup T_p^1$  and  $T_r^0 \cup T_r^1$  respectively.

**Example 2.4.11.** Let  $T_0$  be the term deduction system with  $\Sigma_0$  of our running example and with the rules of table 5. Let  $T_1$  be the term deduction system with  $\Sigma_1$  of the running example and with deduction rules that can be found in table 9. The sum  $T_0 \oplus T_1$  is defined and is the operational semantics of the theory comprising basic process algebra and projections: BPA + PR.

Next, we define what we will call operational conservativity. This definition is taken from [Verhoef, 1994b], but this notion is already defined by [Groote and Vaandrager, 1992] for the case without extra predicates on states.

**Definition 2.4.12.** Let  $T^i = (\Sigma_i, D_i)$  be term deduction systems ( $i = 0, 1$ ) with  $T = (\Sigma, D) := T^0 \oplus T^1$  defined. Let  $D = D(T_p, T_r)$ . The term deduction system  $T$  is called an operationally conservative extension of  $T^0$  if for all  $s, u \in C(\Sigma_0)$ , for all relation symbols  $R \in T_r$  and predicate symbols  $P \in T_p$ , and for all  $t \in C(\Sigma)$  we have

$$T \vdash sRt \iff T^0 \vdash sRt$$

and

$$T \vdash Pu \iff T^0 \vdash Pu.$$

Before we can continue with a theorem that gives sufficient conditions when a term deduction system is an operationally conservative extension of another such system, we need one more definition. This definition originates from [Groote and Vaandrager, 1992].

**Definition 2.4.13.** Let  $T = (\Sigma, D)$  be a term deduction system and let  $F$  be a set of formulas. The variable dependency graph of  $F$  is a directed graph with variables occurring in  $F$  as its nodes. The edge  $x \longrightarrow y$  is an edge of the variable dependency graph if and only if there is a relation  $tRs \in F$  with  $x \in \text{var}(t)$  and  $y \in \text{var}(s)$ .

The set  $F$  is called well-founded if any backward chain of edges in its variable dependency graph is finite. A deduction rule is called well-founded if its set of hypotheses is so. A term deduction system is called well-founded if all its deduction rules are well-founded.

**Example 2.4.14.** Definition 2.4.13 expresses that a rule is well-founded if the set of premises does not contain cyclic references to variables (in case this set is finite). So, for instance, if there is a premise  $xRx$  then there is a cyclic reference to the variable  $x$ . Also the two premises  $xRy$  and  $ySx$  comprise a cyclic reference to  $x$ . Since we do not have such premises in the operational semantics of BPA, it is not hard to verify that the deduction rules of table 5 are well-founded.

Now, we are in a position to state a theorem providing us with sufficient conditions so that a term deduction system is an operationally conservative extension of another one. This theorem is based on a more general theorem of [Verhoef, 1994b]. A more restrictive version of this theorem was first formulated by [Groote and Vaandrager, 1992].

**Theorem 2.4.15.** *Let  $T^0 = (\Sigma_0, D_0)$  be a pure well-founded term deduction system in path format. Let  $T^1 = (\Sigma_1, D_1)$  be a term deduction system in path format. If there is a conclusion  $sRt$  or  $Ps$  of a rule  $d_1 \in D_1$  with  $s = x$  or  $s = f(x_1, \dots, x_n)$  for an  $f \in \Sigma_0$ , we additionally require that  $d_1$  is pure, well-founded,  $t \in O(\Sigma_0)$  for premises  $tRy$  of  $d_1$ , and that there is a premise containing only  $\Sigma_0$  terms and a new relation or predicate symbol. Now if  $T = T^0 \oplus T^1$  is defined then  $T$  is an operationally conservative extension of  $T_0$ .*

**Example 2.4.16.** We already gave the definition of a pure term deduction system in definition 2.2.30. It is not hard to see that the term deduction system  $T_0$  of our running example is pure. It is also not difficult to see that  $T_1$  of our running example is in *path* format. Moreover, since there is no deduction rule in  $T_1$  with an old function symbol or a variable on the vital position, we do not need to check the additional requirements. So, since the sum is defined we conclude with the above theorem that  $T_0 \oplus T_1$  is an operationally conservative extension of  $T_0$ .

Now that we have the operational conservativity, we need to make the connection with the usual conservativity. Following [Verhoef, 1994b], henceforth we will call this well-known property equational conservativity to exclude possible confusion with the already introduced notion of operational conservativity. As an intermediate step, we will first define the notion of operational conservativity up to  $\varphi$  equivalence. Here,  $\varphi$  equivalence is some (semantical) equivalence that is defined in terms of relation and predicate symbols only. Strong bisimulation equivalence is an example of an equivalence that is definable exclusively in terms of relation and predicate symbols. This definition was first formulated by [Groote and Vaandrager, 1992] for the case of operational conservativity up to strong bisimulation equivalence. Roughly, if original terms  $s$  and  $t$  are bisimilar in the extended system, if and only if they are bisimilar in the original system we call the large system a conservative extension up to bisimulation equivalence of the



original one. The next definition expresses this for any equivalence.

**Definition 2.4.17.** Let  $T^i = (\Sigma_i, D_i)$  be term deduction systems ( $i = 0, 1$ ) with  $T = (\Sigma, D) := T^0 \oplus T^1$  defined. If we have for all  $s, t \in C(\Sigma_0)$

$$s =_{\varphi}^{\oplus} t \iff s =_{\varphi}^0 t$$

we say that  $T$  is an operationally conservative extension of  $T_0$  up to  $\varphi$  equivalence, where  $\varphi$  is some semantical equivalence that is defined in terms of relation and predicate symbols only. By  $s =_{\varphi} t$  we mean that  $s$  and  $t$  are in the same  $\varphi$  equivalence class. The superscripts  $\oplus$  and  $0$  are to express the system in which this holds.

**Remark 2.4.18.** Many equivalences are definable in terms of relation and predicate symbols only: for instance, trace equivalence, completed trace equivalence, failure equivalence, readiness equivalence, failure trace equivalence, ready trace equivalence, possible future equivalence, simulation equivalence, complete simulation equivalence, ready simulation equivalence, nested simulation equivalence, strong bisimulation equivalence, weak bisimulation equivalence,  $\eta$  bisimulation equivalence, delay bisimulation equivalence, branching bisimulation equivalence, and more equivalences. We refer to Glabbeek's linear time – branching time spectra [Glabbeek, 1990; Glabbeek, 1993] for more information on these equivalences.

Next, we formulate a theorem stating that if a large system is an operationally conservative extension of a small system, then it is also an operationally conservative extension up to any equivalence that is definable in terms of relation and predicate symbols only. This theorem is taken from [Verhoef, 1994b]. This theorem was formulated by [Groote and Vaandrager, 1992] for the case of strong bisimulation equivalence.

**Theorem 2.4.19.** Let  $T^i = (\Sigma_i, D_i)$  be term deduction systems ( $i = 0, 1$ ) and let  $T = T^0 \oplus T^1$  be defined. If  $T$  is an operationally conservative extension of  $T_0$  then it is also an operationally conservative extension up to  $\varphi$  equivalence, where  $\varphi$  is an equivalence relation defined exclusively in terms of predicate and relation symbols.

**Example 2.4.20.** For our running example it will be clear that the term deduction system of the sum  $T_0 \oplus T_1$  is an operationally conservative extension up to strong bisimulation equivalence of the base system  $T_0$ .

Now that we have the intermediate notion of operational conservativity up to some equivalence, we will come to the well-known notion that in this chapter we will call equational conservativity. We recall that an equational specification is a pair consisting of a signature and a set of equations over this signature. First we define how we combine equational specifications into larger ones.

**Definition 2.4.21.** Let  $L_i = (\Sigma_i, E_i)$  be equational specifications ( $i =$



0, 1). Let  $\Sigma_0 \oplus \Sigma_1$  be defined. Then the sum  $L_0 \oplus L_1$  of  $L_0$  and  $L_1$  is the equational specification  $(\Sigma_0 \oplus \Sigma_1, E_0 \cup E_1)$ .

**Example 2.4.22.** Let  $L_0$  be the equational specification that consists of the signature  $\Sigma_0$  of our running example and the equations  $E_0$  of BPA that we already listed in table 1: the axioms A1–A5. Let  $L_1$  be the equational specification with as signature  $\Sigma_1$  of our running example and with equations that we presented in table 7: PR1–PR4. Now the sum  $L_0 \oplus L_1$  is defined and equals the equational specification that we baptized BPA + PR.

Next, we define the notion of equational conservativity.

**Definition 2.4.23.** Let  $L_i = (\Sigma_i, E_i)$  be equational specifications ( $i = 0, 1$ ) and let  $L = L_0 \oplus L_1$  be defined. We say that  $L$  is an equationally conservative extension, or simply a conservative extension of  $L_0$ , if for all  $s, t \in C(\Sigma_0)$

$$L \vdash s = t \iff L_0 \vdash s = t.$$

Here,  $\vdash$  stands for derivability in equational logic see, e.g. [Klop, 1992].

Now we have all the prerequisites to formulate the equational conservativity theorem. This theorem is taken from [Verhoef, 1994b].

**Theorem 2.4.24.** Let  $L_i = (\Sigma_i, E_i)$  be equational specifications ( $i = 0, 1$ ) and let  $L = (\Sigma, E) = L_0 \oplus L_1$  be defined. Let  $T_i = (\Sigma_i, D_i)$  be term deduction systems and let  $T = T_0 \oplus T_1$ . Let  $\varphi$  be an equivalence that is definable in terms of predicate and relation symbols only. Let  $E_0$  be a complete axiomatization with respect to the  $\varphi$  equivalence model induced by  $T_0$  and let  $E$  be a sound axiomatization with respect to the  $\varphi$  equivalence model induced by  $T$ . If  $T$  is an operationally conservative extension of  $T_0$  up to  $\varphi$  equivalence then  $L$  is an equationally conservative extension of  $L_0$ .

Now, we can apply the equational conservativity theorem to prove the conservativity of BPA + PR over BPA.

**Theorem 2.4.25.** If  $t$  and  $s$  are closed BPA terms, then we have

$$\text{BPA} \vdash t = s \iff \text{BPA} + \text{PR} \vdash t = s.$$

**Proof.** On the way to this proof we checked all the conditions of the theorem in the example paragraphs except for the soundness and completeness of BPA and the soundness of BPA + PR. Fortunately, we already proved these conditions. The soundness and completeness of BPA is proved in theorems 2.2.33 and 2.2.35 respectively and the soundness of BPA + PR is proved in theorem 2.4.7. So, we can apply theorem 2.4.24 and conclude that BPA + PR is an equationally conservative extension of BPA. ■

Now that we have the conservativity of BPA + PR, we can immediately prove its completeness from the completeness of the subsystem BPA. We

will not do this directly, but we will formulate a general completeness theorem that can be found in [Verhoef, 1994b]; it is a simple corollary of the equational conservativity theorem. This completeness theorem states that the combination of conservativity, elimination of extra operators, and the completeness of the subsystem yields the completeness of the extension. For the formulation of the next theorem, we stick to the notations and assumptions stated in theorem 2.4.24.

**Theorem 2.4.26.** *If in addition to the conditions of theorem 2.4.24 the equational specification  $L$  has the elimination property for  $L_0$  (see definition 2.4.3) then we have that  $E$  is a complete axiomatization with respect to the  $\varphi$  equivalence model induced by the term deduction system  $T$ .*

**Theorem 2.4.27.** *The axiom system  $\text{BPA} + \text{PR}$  is a complete axiomatization of the set of closed  $\text{BPA} + \text{PR}$  terms modulo bisimulation equivalence.*

**Proof.** We apply theorem 2.4.26. We already know that the conditions of the conservativity theorem are satisfied. So we only need to check the additional one. According to theorem 2.4.4 the elimination condition is satisfied. So the conditions of theorem 2.4.26 are satisfied and we are done. ■

## 2.4.2 Recursion and projection

In this subsection we will add recursion and projections to the theory  $\text{BPA}$ .

We will define the equational specification  $\text{BPAREC} + \text{PR}$  by means of a combination of  $\text{BPAREC}$  and  $\text{BPA} + \text{PR}$ .

The signature of  $\text{BPAREC} + \text{PR}$  consists of the signature of  $\text{BPAREC}$  plus for each  $n \geq 1$  a unary function  $\pi_n$  (projections). The axioms of  $\text{BPAREC} + \text{PR}$  are the axioms of  $\text{BPAREC}$  and the axioms of table 7.

*Proof rule* We will discuss a proof rule expressing that a process that can be specified with the aid of a guarded recursive specification is determined by its finite projections. This rule is a restricted version of the rule that is defined in definition 2.4.1 and is also due to [Bergstra and Klop, 1986].

**Definition 2.4.28.** Let  $x$  and  $y$  be processes such that  $x$  or  $y$  (or both) can be specified with the aid of a guarded recursive specification. The *restricted approximation induction principle* ( $\text{AIP}^-$ ) is the following assumption. If for all  $n \geq 1$  we have  $\pi_n(x) = \pi_n(y)$  then  $x = y$ .

**Remark 2.4.29.** The principle  $\text{AIP}^-$  is more restrictive than necessary. A more general approximation induction principle is defined in [Glabbeek, 1987].

**Theorem 2.4.30.** *Let  $x$  be a solution of a guarded recursive specification. Then  $\pi_n(x)$  can be rewritten into a closed  $\text{BPA}$  term for all  $n \geq 1$ .*

**Proof.** Without loss of generality we may assume that  $x$  is a solution of a completely guarded recursive specification. Let the right-hand side of the

recursion equation of  $x$  be called  $s_1$ . Suppose that we have defined  $s_n$ . Then we obtain  $s_{n+1}$  as follows: substitute for each variable in  $s_n$  the right-hand side of its recursion equation. It is not hard to see that for every  $n \geq 1$  we have  $\pi_n(x) = \pi_n(s_n)$  and that the latter term can be rewritten into a closed BPA term. ■

The following corollary is called *the projection theorem*.

**Corollary 2.4.31.** *Suppose that we have two solutions  $x_1$  and  $x_2$  of a guarded recursive specification belonging to the same recursion equation. Then for all  $n \geq 1$  :  $\pi_n(x_1) = \pi_n(x_2)$ .*

**Proof.** This follows immediately from the proof of theorem 2.4.30. ■

**Theorem 2.4.32.** *The principle  $\text{AIP}^-$  implies the principle RSP.*

**Proof.** This follows immediately from corollary 2.4.31. ■

*Semantics* The semantics of  $\text{BParec} + \text{PR}$  is given by means of a term deduction system  $T(\text{BParec} + \text{PR})$  with as signature the signature of  $\text{BParec} + \text{PR}$  and as rules the rules in tables 5, 6, and 9. Bisimulation equivalence is a congruence on the structured state system  $L(\text{BParec} + \text{PR})$  induced by  $T(\text{BParec} + \text{PR})$ ; see 2.2.31. So the quotient of closed  $\text{BParec} + \text{PR}$  terms modulo strong bisimulation equivalence is well-defined; that is, the operators of  $\text{BParec} + \text{PR}$  can be defined on this quotient by taking representatives. The next theorem states that this quotient is a model of  $\text{BParec} + \text{PR}$ .

**Theorem 2.4.33.** *The set of closed  $\text{BParec} + \text{PR}$  terms modulo bisimulation equivalence is a model of  $\text{BParec} + \text{PR}$ .*

**Proof.** Since strong bisimulation equivalence is a congruence, we only need to verify the soundness of each axiom. This has been done in the proofs of theorems 2.3.13 and 2.4.7. ■

**Theorem 2.4.34.** *The model of closed  $\text{BParec} + \text{PR}$  terms modulo bisimulation equivalence satisfies RDP,  $\text{AIP}^-$ , and RSP.*

**Proof.** The principle RDP is proved as in theorem 2.2.33. With the aid of theorem 2.4.32 it suffices to prove that the model satisfies  $\text{AIP}^-$ . This proof is taken from [Glabbeek, 1987].

So let  $x$  and  $y$  be closed  $\text{BParec} + \text{PR}$  terms such that for all  $n \geq 1$  we have  $\pi_n(x) \sim \pi_n(y)$  (the symbol  $\sim$  stands for the bisimulation relation). Suppose that  $y$  can be specified with the aid of a guarded recursive specification. We have to prove that  $x$  and  $y$  are bisimilar. We relate  $u$  and  $v$ , notation  $u R v$ , if and only if  $v$  can be specified with the aid of a guarded recursive specification and if for all  $n \geq 1$  :  $\pi_n(u) \sim \pi_n(v)$ . Note that  $x R y$ . We will prove that  $R$  is a bisimulation relation. So we have to distinguish three cases.

Suppose that  $u R v$  and  $u \xrightarrow{a} u'$ . Define for  $n \geq 1$

$$S_n = \{v^* \mid v \xrightarrow{a} v^*, \pi_n(u') \sim \pi_n(v^*)\}.$$

To prove that there is a  $v'$  with  $v \xrightarrow{a} v'$  and  $u' R v'$  it suffices to show that the intersection of the  $S_n$  contains an element. Firstly, every  $S_n$  contains an element since  $\pi_{n+1}(u) \sim \pi_{n+1}(v)$ . Secondly, each  $S_n$  is finite. For  $v$  can be specified with the aid of a guarded recursive specification, so there is a completely guarded term  $t$  with  $v = t$ . We can rewrite this term using the rewrite rules RA3, RA4, and RA5 of table 8 to a sum of terms. The summands are either atomic actions or products  $t' \cdot t''$  and  $t'$  is not a sum or a product. Since  $t$  is completely guarded it must be an atomic action. So  $v$  has the form

$$v = \sum_{i=1}^n a_i \cdot v_i + \sum_{j=1}^m b_j.$$

This means that the set  $S_n$  is finite. Thirdly, we have  $S_{n+1} \subseteq S_n$  for all  $n \geq 1$ , since  $\pi_{n+1}(u') \sim \pi_{n+1}(v^*)$  implies  $\pi_n(u') \sim \pi_n(v^*)$ . From these three observations we can conclude that the sequence  $S_1, S_2, \dots$  must remain constant from some index onwards. Thus, the intersection of the  $S_n$  is not empty.

Now suppose  $u R v$  and  $v \xrightarrow{a} v'$ . Define, as above, for all  $n \geq 1$

$$S_n = \{u^* \mid u \xrightarrow{a} u^*, \pi_n(u^*) \sim \pi_n(v')\}.$$

We can again observe that every  $S_n$  contains an element and that the sequence  $S_1, S_2, \dots$  is decreasing (but not that each  $S_n$  is finite). So we can choose for each  $n \geq 1$  an element  $u_n \in S_n$ . By the first part of the proof we know that there are  $v_n$  with  $v \xrightarrow{a} v_n$  and  $u_n R v_n$ . But since  $v$  can be specified with the aid of a guarded recursive specification there is a  $v^*$  that occurs infinitely many times in the sequence  $v_1, v_2, \dots$ . Let  $v^* = v_k$  for some index  $k$ . We show that  $u_k R v'$ , which proves the second case. So fix an  $n \geq 1$ . Then there is an index  $m > n$  with  $v^* = v_m$  and we find  $\pi_n(u_m) \sim \pi_n(v')$ , since  $u_m \in S_m \subseteq S_n$ . Moreover, we have  $u_k R v^*$  and  $u_m R v^*$ . So  $\pi_n(u_k) \sim \pi_n(u_m)$  and we find  $\pi_n(u_k) \sim \pi_n(v')$ . So we have  $u_k R v'$ , since  $n \geq 1$  was arbitrarily chosen.

Finally, note that if  $u R v$  then we have

$$u \xrightarrow{a} \surd \iff \pi_2(u) \xrightarrow{a} \surd \iff \pi_2(v) \xrightarrow{a} \surd \iff v \xrightarrow{a} \surd.$$

So  $R$  is a bisimulation. This finishes the proof. ■

**Theorem 2.4.35.** *The principle AIP does not hold in the model of closed BParec + PR terms modulo bisimulation equivalence.*



**Proof.** Consider the two recursive specifications that we defined in example 2.3.2. Let  $x = \langle X \mid E_1 \rangle$  and  $y = \langle Y \mid E_2 \rangle$ . With the aid of figure 5 we can see that for all  $n \geq 1$  we have that  $\pi_n(x+y) \sim \pi_n(x)$  but that  $x+y \not\sim x$ . Note that  $E_1$  is not guarded so  $x+y$  and  $x$  are not specified with the aid of a guarded recursive specification. ■

The following theorem concerns the theory  $\text{BPAREC}$ . The result was already stated in theorem 2.3.13. We postponed the proof of this until now, since we want to use the fact that  $\text{RSP}$  holds in  $\text{BPAREC} + \text{PR}$ .

**Theorem 2.4.36.** *The model of closed  $\text{BPAREC}$  terms modulo bisimulation equivalence satisfies  $\text{RSP}$ .*

**Proof.** Let  $E(V)$  be a guarded recursive specification and suppose that we have two solutions, say  $x$  and  $y$ , belonging to the same recursion equation. We have to show that  $x \sim y$ . Since  $x$  and  $y$  are also  $\text{BPAREC} + \text{PR}$  terms we find with the aid of theorem 2.4.34 that  $x \sim y$ , which proves the theorem. ■

## 2.5 Deadlock

Usually deadlock stands for a process that has stopped its executions and cannot proceed. In this subsection we will extend the theory  $\text{BPA}$  with a process named *deadlock*. We can distinguish between successful and unsuccessful termination in the presence of deadlock. This subsection is based on [Bergstra and Klop, 1984a; Bergstra and Klop, 1995].

Let  $x \cdot y$  be a sequential composition of two processes. The process  $y$  starts if  $x$  has terminated. But if  $x$  reaches a state of inaction due to deadlock, we do not want  $y$  to start: we want it only to start when  $x$  terminates successfully. We will axiomatize the behaviour of a deadlocked process called  $\delta$  in table 10.

The signature of the equational specification  $\text{BPA}_\delta$  is the signature of  $\text{BPA}$  extended with a constant  $\delta \notin A$  called deadlock, or inaction. The axioms of the equational specification  $\text{BPA}_\delta$  are the axioms of  $\text{BPA}$  in table 1 plus the two axioms in table 10 (A6 and A7). We will discuss them now.

Equation A6 expresses that  $\delta$  is a neutral element with respect to the alternative composition; it says that no deadlock will occur as long as there is an alternative that can proceed. Axiom A7 says that the constant  $\delta$  is a left-zero element for the sequential composition. It says that after a deadlock has occurred, no other actions can possibly follow. Actually, *inaction* would be a better name for the constant  $\delta$ , for a process  $a + \delta$  ( $a \in A$ ) contains no deadlock. Deadlock only occurs if there is no alternative to  $\delta$ , as in  $a \cdot \delta$ .

So using the process  $\delta$  we can distinguish between successful and unsuccessful termination. Thus, the process  $a\delta$  terminates unsuccessfully whereas  $a$  terminates successfully.



*Structural induction* In  $\text{BPA}_\delta$  we can use the technique of structural induction just like in BPA (compare page 155). We will adjust the definition of a basic term (see definition 2.2.3) and we will mention the result that every closed term over  $\text{BPA}_\delta$  can be written as a basic term.

**Definition 2.5.1.** The constant  $\delta$  is a basic term over  $\text{BPA}_\delta$ . An atomic action is a basic term. If  $t$  is a basic term and  $a \in A$ , then  $a \cdot t$  is a basic term. If  $t$  and  $s$  are basic terms, then  $t + s$  is a basic term.

We recall that a closed term over  $\text{BPA}_\delta$  is a  $\text{BPA}_\delta$  term without variables.

**Remark 2.5.2.** If we consider terms  $t$  and  $s$  to be identical if

$$A1, A2 \vdash t = s,$$

we can see that basic terms have the following form:

$$\sum_{i=1}^n a_i \cdot t_i + \sum_{j=1}^m b_j,$$

where  $a_i \in A$ ,  $b_j \in A \cup \{\delta\}$ ,  $1 \leq i \leq n$ ,  $1 \leq j \leq m$ , and  $n + m \geq 1$ .

**Proposition 2.5.3.** Let  $t$  be a closed  $\text{BPA}_\delta$  term. Then there is a basic term  $s$  such that  $\text{BPA}_\delta \vdash t = s$ .

**Proof.** The proof of this proposition can be given along the same lines as the proof of proposition 2.2.5. ■

*Semantics* As usual, we give the semantics by means of a term deduction system. Take for the signature of  $T(\text{BPA}_\delta)$  the signature of  $\text{BPA}_\delta$  and for the set of rules just the ones of BPA of table 5. Since bisimulation equivalence is a congruence (2.2.31), the quotient of closed  $\text{BPA}_\delta$  terms modulo bisimulation equivalence is well-defined so the operators of  $\text{BPA}_\delta$  can be defined on this quotient using representatives. This quotient is a model of  $\text{BPA}_\delta$ .

**Theorem 2.5.4.** The set of closed  $\text{BPA}_\delta$  terms modulo bisimulation equivalence is a model of  $\text{BPA}_\delta$ .

**Proof.** It suffices to check the soundness of each axiom, since bisimulation equivalence is a congruence. Axioms A1–A5 are treated as in theorem 2.2.33 since there are no transitions for  $\delta$ . Axiom A6 is treated as A1. For axiom A7 take the relation that only relates  $\delta x$  and  $\delta$ . ■

**Table 10.** Deadlock.

$x + \delta = x$	A6
$\delta x = \delta$	A7

**Theorem 2.5.5.** *The axiom system  $\text{BPA}_\delta$  is a complete axiomatization of the set of closed  $\text{BPA}_\delta$  terms modulo bisimulation equivalence.*

**Proof.** Since there are no new transitions for the constant  $\delta$ , this is proved as theorem 2.2.35. ■

### 2.5.1 Extensions of $\text{BPA}_\delta$

In this subsection we will discuss the extensions of  $\text{BPA}_\delta$  with recursion and/or projections.

*Recursion* We can add recursion to  $\text{BPA}_\delta$  in exactly the same way as we did for BPA. The equational specification  $\text{BPA}_\delta\text{rec}$  contains the signature of  $\text{BPArec}$  and a constant  $\delta \notin A$ . The axioms of  $\text{BPA}_\delta\text{rec}$  are the axioms of  $\text{BPArec}$  plus the axioms of table 10.

Note that  $\delta \notin A$  so it cannot serve as a guard:  $\delta X$  is not completely guarded but it is guarded since  $\delta X = \delta$ .

The semantics of  $\text{BPA}_\delta\text{rec}$  can be given by means of the term deduction system  $T(\text{BPA}_\delta\text{rec})$  that has as its signature the signature of  $\text{BPA}_\delta\text{rec}$  and as rules the rules of tables 5 and 6. Since bisimulation equivalence is a congruence (2.2.31), the operators of  $\text{BPA}_\delta\text{rec}$  can be defined on the quotient algebra of closed  $\text{BPA}_\delta\text{rec}$  terms with respect to bisimulation equivalence. This quotient is a model of  $\text{BPA}_\delta\text{rec}$  and it satisfies RDP. To prove this, combine the proofs of theorems 2.3.13 and 2.5.4. Moreover this model satisfies RSP, which can be proved when we combine  $\text{BPA}_\delta\text{rec}$  with projections.

*Projection* We can extend  $\text{BPA}_\delta$  with projections in exactly the same way as we did for BPA. The equational specification  $\text{BPA}_\delta + \text{PR}$  has as its signature the signature of  $\text{BPA} + \text{PR}$  and a constant  $\delta \notin A$ . Its axioms are the axioms of  $\text{BPA} + \text{PR}$  and the ones concerning deadlock (tables 1, 7, and 10). We assume that  $a$  ranges over  $A \cup \{\delta\}$  in table 7 on projections.

The following theorem states that projection operators can be eliminated from closed terms.

**Theorem 2.5.6.** *For every closed  $\text{BPA}_\delta + \text{PR}$  term  $t$  there is a basic  $\text{BPA}_\delta$  term  $s$  such that  $\text{BPA}_\delta + \text{PR} \vdash t = s$ .*

**Proof.** Use the term rewriting system that consists of the rules in table 8 and in addition the rewrite rule  $\delta \cdot x \rightarrow \delta$  and show that this term rewriting system is terminating. Hint: use theorem 2.2.17. The rest of the proof is straightforward and therefore omitted. ■

**Proposition 2.5.7.** *Let  $t$  be a closed  $\text{BPA}_\delta + \text{PR}$  term. Then there is an  $n \geq 1$  such that for all  $k \geq n$  we have  $\text{BPA}_\delta + \text{PR} \vdash \pi_k(t) = t$ .*

**Proof.** With theorem 2.5.6 it suffices to prove the proposition for basic  $\text{BPA}_\delta$  terms  $t$ . So we can use structural induction on  $t$  to prove the proposition. ■

The semantics of  $\text{BPA}_\delta + \text{PR}$  can be given with the term deduction system  $T(\text{BPA}_\delta + \text{PR})$  that has as its signature the signature of  $\text{BPA}_\delta + \text{PR}$  and as rules the rules of tables 5 and 9. Since bisimulation equivalence is a congruence (2.2.31), it follows that the operators of  $\text{BPA}_\delta + \text{PR}$  can be defined on the quotient algebra of closed  $\text{BPA}_\delta + \text{PR}$  terms with respect to bisimulation equivalence. This quotient is a model of  $\text{BPA}_\delta + \text{PR}$  and it satisfies AIP. To prove this, combine the proofs of theorems 2.4.7 and 2.5.4. Moreover, according to theorem 2.4.24 we have that  $\text{BPA}_\delta + \text{PR}$  is a conservative extension of  $\text{BPA}_\delta$ . So with theorem 2.4.26 we find that  $\text{BPA}_\delta + \text{PR}$  is a complete axiomatization of this model (use also theorem 2.5.6).

*Recursion and projection* Here we extend  $\text{BPA}_\delta$  with recursion and projections. The equational specification  $\text{BPA}_\delta\text{rec} + \text{PR}$  has as its signature the signature of  $\text{BPA}_\delta\text{rec}$  plus for each  $n \geq 1$  a unary function  $\pi_n$ . The axioms are the axioms of  $\text{BPA}_\delta\text{rec}$  and the axioms concerning projections (table 7). We assume that  $a$  ranges over  $A \cup \{\delta\}$  in this table.

The standard facts (and their proofs) of subsection 2.4.2 are easily transposed to the present situation. Their translation is, in short: every projection of a solution of a guarded recursive specification can be rewritten into a closed  $\text{BPA}_\delta$  term; the projection theorem 2.4.31 holds; and  $\text{AIP}^-$  implies RSP.

The semantics of  $\text{BPA}_\delta\text{rec} + \text{PR}$  is given by means of a term deduction system  $T(\text{BPA}_\delta\text{rec} + \text{PR})$ . Its signature is the signature of  $\text{BPA}_\delta\text{rec} + \text{PR}$  and its rules are those of  $T(\text{BPA}_\delta + \text{PR})$  plus the rules concerning recursion that are presented in table 6. Since bisimulation equivalence is a congruence, the quotient algebra of closed  $\text{BPA}_\delta\text{rec} + \text{PR}$  terms with respect to bisimulation equivalence is well-defined, and the operators of  $\text{BPA}_\delta\text{rec} + \text{PR}$  can be defined on this quotient. This quotient is a model of  $\text{BPA}_\delta\text{rec} + \text{PR}$  and it satisfies RDP, RSP, and  $\text{AIP}^-$ , but not its unrestricted version AIP. We can prove that  $\text{BPA}_\delta\text{rec}$  satisfies RSP. This is proved in the same way as theorem 2.4.36.

## 2.6 Empty process

In many situations it is useful to have a constant process that stands for immediate successful termination. In this subsection we will extend the equational specifications  $\text{BPA}$  and  $\text{BPA}_\delta$  with a process that is only capable of terminating successfully. We will call such a process the *empty process* and we will denote it by  $\epsilon$ . This constant originates from [Koymans and Vrancken, 1985]. Another reference to this constant is [Vrancken, 1986].

The empty process is a counterpart of the deadlock process. The deadlock process stands for immediate *unsuccessful* termination while the empty process stands for immediate *successful* termination. Moreover, the combination of the two axioms A8 and A9 of table 11 express that  $\epsilon$  is a neutral element with respect to the *sequential* composition whereas axioms A1

and A6 express that  $\delta$  is a neutral element with respect to the *alternative* composition. Note that successful termination (not in a sum context) after the execution of at least one action can already be expressed in systems without  $\varepsilon$ , as  $a \cdot \varepsilon = a$  ( $a \in A$ ).

The equational specifications  $\text{BPA}_\varepsilon$  and  $\text{BPA}_{\delta\varepsilon}$  are defined as follows.

The signature of  $\text{BPA}_\varepsilon$  consists of the signature of BPA extended with a constant  $\varepsilon \notin A$  called the empty process. The equations of  $\text{BPA}_\varepsilon$  are the axioms of BPA and the axioms A8 and A9 of table 11.

The signature of  $\text{BPA}_{\delta\varepsilon}$  consists of the signature of  $\text{BPA}_\delta$  extended with a constant  $\varepsilon \notin A \cup \{\delta\}$ . The axioms of  $\text{BPA}_{\delta\varepsilon}$  are the ones of  $\text{BPA}_\delta$  plus A8 and A9.

*Structural induction* In  $\text{BPA}_\varepsilon$  and  $\text{BPA}_{\delta\varepsilon}$  we can use the technique of structural induction just like in BPA or  $\text{BPA}_\delta$  since every closed term can be written as a basic term. We will adjust the definition of a basic term to the present situation and we will mention that closed terms over  $\text{BPA}_\varepsilon$  or  $\text{BPA}_{\delta\varepsilon}$  can be written as basic terms.

**Definition 2.6.1.** A basic term over  $\text{BPA}_\varepsilon$  is defined as follows.

An atomic action is a basic term over  $\text{BPA}_\varepsilon$ . The constant  $\varepsilon$  is a basic term over  $\text{BPA}_\varepsilon$ . If  $t$  is a basic term over  $\text{BPA}_\varepsilon$  and  $a \in A$ , then  $a \cdot t$  is a basic term over  $\text{BPA}_\varepsilon$ . If  $t$  and  $s$  are basic terms over  $\text{BPA}_\varepsilon$ , then  $t + s$  is a basic term over  $\text{BPA}_\varepsilon$ .

A basic term over  $\text{BPA}_{\delta\varepsilon}$  is defined as follows.

An atomic action is a basic term over  $\text{BPA}_{\delta\varepsilon}$ . The constants  $\delta$  and  $\varepsilon$  are basic terms over  $\text{BPA}_{\delta\varepsilon}$ . If  $t$  is a basic term over  $\text{BPA}_{\delta\varepsilon}$  and  $a \in A$ , then  $a \cdot t$  is a basic term over  $\text{BPA}_{\delta\varepsilon}$ . If  $t$  and  $s$  are basic terms over  $\text{BPA}_{\delta\varepsilon}$ , then  $t + s$  is a basic term over  $\text{BPA}_{\delta\varepsilon}$ .

We recall that a closed term over  $\text{BPA}_\varepsilon/\text{BPA}_{\delta\varepsilon}$  is a  $\text{BPA}_\varepsilon/\text{BPA}_{\delta\varepsilon}$  term without variables.

**Remark 2.6.2.** If we consider terms identical that only differ in the order of the summands, basic terms over  $\text{BPA}_\varepsilon$  or  $\text{BPA}_{\delta\varepsilon}$  are of the form

$$\sum_{i=1}^n a_i \cdot t_i + \sum_{j=1}^m b_j,$$

where  $a_i \in A$ ,  $b_j \in A \cup \{\delta, \varepsilon\}$ ,  $1 \leq i \leq n$ ,  $1 \leq j \leq m$ , and  $n + m \geq 1$ .

**Proposition 2.6.3.** Let  $t$  be a closed  $\text{BPA}_\varepsilon/\text{BPA}_{\delta\varepsilon}$  term. Then there is a

Table 11. Empty process.

$x\varepsilon = x$	A8
$\varepsilon x = x$	A9



basic term  $s$  such that  $\text{BPA}_\varepsilon/\text{BPA}_{\delta\varepsilon} \vdash t = s$ .

**Proof.** The proof of this proposition can be given along the same lines as the proof of proposition 2.2.5. ■

*Semantics* We give the semantics of  $\text{BPA}_\varepsilon$  and  $\text{BPA}_{\delta\varepsilon}$  by means of term deduction systems. We handle both cases at the same time. Take for the signature of  $T(\text{BPA}_{(\delta)\varepsilon})$  the signature of  $\text{BPA}_{(\delta)\varepsilon}$  and for the set of rules the ones that are presented in table 12. This operational semantics is taken from [Baeten and Glabbeek, 1987].

The term deduction systems that we consider here differ from the ones that we treated before: instead of successful termination predicates  $\cdot \xrightarrow{a} \checkmark$  we now have a termination option predicate; it is denoted postfix:  $\cdot \downarrow$ . Since both are unary predicates on states we can still use the general theory on structured operational semantics that we treated in subsection 2.2.3. In particular we can use theorem 2.2.31 to prove that bisimulation equivalence is a congruence. So, the quotient algebra of closed  $\text{BPA}_{(\delta)\varepsilon}$  terms with respect to bisimulation equivalence is well-defined for the operators of  $\text{BPA}_{(\delta)\varepsilon}$ . This quotient is a model of  $\text{BPA}_{(\delta)\varepsilon}$ .

**Theorem 2.6.4.** *The set of closed  $\text{BPA}_{(\delta)\varepsilon}$  terms modulo bisimulation equivalence is a model of  $\text{BPA}_{(\delta)\varepsilon}$ .*

**Proof.** Easy. A1–A7 are treated as usual, A8 as A5, and A9 as A1. ■

**Theorem 2.6.5.** *The axiom system  $\text{BPA}_{(\delta)\varepsilon}$  is a complete axiomatization of the set of closed  $\text{BPA}_{(\delta)\varepsilon}$  terms modulo bisimulation equivalence.*

**Proof.** This is proved along the same lines as theorem 2.2.35 if we rephrase lemma 2.2.34 as follows. First, redefine the function  $n$ :  $n(\varepsilon) = n(\delta) = 1$ ,  $n(a) = 2$  for all  $a \in A$ , and  $n(x + y) = n(xy) = n(x) + n(y)$ . Secondly, replace the first case of 2.2.34 by  $T(\text{BPA}_{(\delta)\varepsilon}) \vdash x \downarrow \implies \text{BPA}_{(\delta)\varepsilon} \vdash x = \varepsilon + x$ . ■

### 2.6.1 Conservativity

In this subsection we will explain how to prove that  $\text{BPA}_\varepsilon$  is a conservative extension of  $\text{BPA}$ . We cannot immediately use the theory of subsec-

**Table 12.** Derivation rules of  $T(\text{BPA}_{(\delta)\varepsilon})$ .

$a \xrightarrow{a} \varepsilon \quad \varepsilon \downarrow$			
$\frac{x \xrightarrow{a} x'}{x + y \xrightarrow{a} x'}$	$\frac{y \xrightarrow{a} y'}{x + y \xrightarrow{a} y'}$	$\frac{x \downarrow}{(x + y) \downarrow}$	$\frac{y \downarrow}{(x + y) \downarrow}$
$\frac{x \xrightarrow{a} x'}{xy \xrightarrow{a} x'y}$	$\frac{x \downarrow, y \xrightarrow{a} y'}{xy \xrightarrow{a} y'}$	$\frac{x \downarrow, y \downarrow}{(xy) \downarrow}$	



tion 2.4.1, since the operational semantics of  $BPA_\varepsilon$  presented in table 12 is not an operationally conservative extension of the operational semantics of BPA that we listed in table 5. For we can prove in the extended system that  $a \xrightarrow{a} \varepsilon$ , whereas in the subsystem we can prove that  $a \xrightarrow{a} \checkmark$ . So we can “reach” a new term if we start with an original term. A possible solution for this problem is to give an alternative operational semantics for  $BPA_\varepsilon$  than the one that we present in table 12.

The special behaviour of the constant  $\varepsilon$  is expressed in the operational semantics of  $BPA_\varepsilon$  as it is presented in table 12. Another possibility is to express the special behaviour of the empty process with the aid of the equivalence relation. A well-known example of this kind is observational congruence due to [Milner, 1980]. There, Milner’s silent action  $\tau$  is treated as a normal atomic action in the operational rules and its special behaviour is expressed with the equivalence relation: observational congruence. In the case of the empty process a similar approach is reported by [Koymans and Vrancken, 1985]. In that paper a graph model was constructed featuring the empty process as an ordinary atomic action. A notion called  $\varepsilon$  bisimulation was defined to express the special behaviour of the empty process. With the approach of [Koymans and Vrancken, 1985] we can use the theory of subsection 2.4.1 to prove the conservativity of  $BPA_\varepsilon$  over BPA. We will sketch the idea and leave the details as an exercise to the interested reader. For the operational rules we just take the operational semantics of BPA where we let  $a$  also range over  $\varepsilon$ . This means that we have, for instance, the rule  $\varepsilon \xrightarrow{\varepsilon} \checkmark$ . Note that this adds a new relation  $\xrightarrow{\varepsilon}$  and a new predicate  $\xrightarrow{\varepsilon} \checkmark$  to the operational rules for BPA. Now with the aid of theorem 2.4.15 it is not hard to see that this term deduction system is an operationally conservative extension of the term deduction system in table 5. By way of an example we will check the conditions of theorem 2.4.15 for one deduction rule in the extended system:

$$\frac{x \xrightarrow{\varepsilon} x'}{x + y \xrightarrow{\varepsilon} x'}.$$

The crucial place to look at is the left-hand side of the conclusion:  $x + y$ . There an original function symbol occurs:  $+$ . Now we need to check that this rule is pure and well-founded. This is easy. Also the terms  $x$  and  $x'$  must be original terms; this is the case since they are variables. And there must be a premise containing only original terms and a new relation or predicate symbol. This is also the case. The other rules are treated equally simply. So, we may apply theorem 2.4.15 and find the operational conservativity. Now this notion termed  $\varepsilon$  bisimulation can be defined exclusively in terms of relation and predicate symbols. So with theorem 2.4.19 we find that the term deduction system belonging to  $BPA_\varepsilon$  is an operationally conservative extension up to  $\varepsilon$  bisimulation equivalence of the term deduction

system belonging to BPA (note that  $\varepsilon$  bisimulation becomes normal strong bisimulation for BPA where no  $\varepsilon$  is present). Now we can apply the equational conservativity theorem 2.4.24 if we know in addition that the model induced by the operational rules modulo  $\varepsilon$  bisimulation equivalence is sound with respect to the axioms of  $\text{BPA}_\varepsilon$  that we listed in tables 1 and 11. This is shown for the graph model by [Koymans and Vrancken, 1985] and this proof transposes effortlessly to the situation with operational rules that we sketched above. This proves that  $\text{BPA}_\varepsilon$  is a conservative extension of BPA.

### 2.6.2 Extensions of $\text{BPA}_{(\delta)\varepsilon}$

In this subsection we will discuss the extensions of  $\text{BPA}_{(\delta)\varepsilon}$  with recursion and/or projections.

*Recursion* We can add recursion to  $\text{BPA}_{(\delta)\varepsilon}$  in exactly the same way as we did for  $\text{BPA}_{(\delta)}$ . The equational specification  $\text{BPA}_{(\delta)\varepsilon}\text{rec}$  contains the signature of  $\text{BPArec}$  and  $(\delta, \varepsilon) \notin A$ . The axioms are the ones of  $\text{BPArec}$  and the axioms of table 11 (and table 10).

Since  $\delta, \varepsilon \notin A$ , they cannot serve as a guard. For instance,  $\varepsilon X$  is neither completely guarded nor guarded.

The semantics of  $\text{BPA}_{(\delta)\varepsilon}\text{rec}$  can be given by means of a term deduction system  $T(\text{BPA}_{(\delta)\varepsilon}\text{rec})$  that has as its signature the signature of  $\text{BPA}_{(\delta)\varepsilon}\text{rec}$  and as its rules the ones of  $T(\text{BPA}_{(\delta)\varepsilon})$  plus the rules of table 13. Since bisimulation equivalence is a congruence (2.2.31), we can define the operators of  $\text{BPA}_{(\delta)\varepsilon}\text{rec}$  on the quotient algebra of closed  $\text{BPA}_{(\delta)\varepsilon}\text{rec}$  terms with respect to bisimulation equivalence. This quotient is a model of  $\text{BPA}_{(\delta)\varepsilon}\text{rec}$  and it satisfies RDP and RSP.

*Projection* We extend the theory  $\text{BPA}_{(\delta)\varepsilon}$  with projections. The equational specification  $\text{BPA}_{(\delta)\varepsilon} + \text{PR}$  has as its signature the one of  $\text{BPA}_{(\delta)} + \text{PR}$  plus a constant  $\varepsilon \notin A$  (and  $\varepsilon \neq \delta$ ). The axioms of  $\text{BPA}_{(\delta)\varepsilon} + \text{PR}$  are the axioms of  $\text{BPA}_{(\delta)} + \text{PR}$  plus the axioms of table 11. Moreover, we assume for axiom PR1 (table 7) that  $a$  may also be  $\varepsilon$ .

The results that we inferred for  $\text{BPA}_{(\delta)} + \text{PR}$  also hold for  $\text{BPA}_{(\delta)\varepsilon} + \text{PR}$ : we can eliminate projections occurring in closed terms and the sequence  $\pi_1(t), \pi_2(t), \dots$  has  $t, t, \dots$  as its tail. We can also prove many conservativity results using subsection 2.4.1. We can, for instance, show that  $\text{BPA}_{(\delta)\varepsilon} + \text{PR}$  is conservative over  $\text{BPA}_{(\delta)\varepsilon}$ . We already showed that  $\text{BPA}_{(\delta)\varepsilon}$  is a conservative extension of BPA, so with transitivity we find that

**Table 13.** Derivation rules for recursion and empty process.

$\frac{\langle s_X \mid E \rangle \downarrow}{\langle X \mid E \rangle \downarrow}$	$\frac{\langle s_X \mid E \rangle \xrightarrow{a} y}{\langle X \mid E \rangle \xrightarrow{a} y}$
-------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------

**Table 14.** Derivation rules for projections with empty process.

$\frac{x \downarrow}{\pi_n(x) \downarrow}$	$\frac{x \xrightarrow{a} x'}{\pi_1(x) \xrightarrow{a} \varepsilon}$	$\frac{x \xrightarrow{a} x'}{\pi_{n+1}(x) \xrightarrow{a} \pi_n(x')}$
--------------------------------------------	---------------------------------------------------------------------	-----------------------------------------------------------------------

$\text{BPA}_{(\delta)\varepsilon} + \text{PR}$  is a conservative extension of BPA. We use the transitivity argument here since the proof that  $\text{BPA}_{(\delta)\varepsilon}$  is a conservative extension of BPA uses another semantics. See subsection 2.6.1 for more information.

The semantics of  $\text{BPA}_{(\delta)\varepsilon} + \text{PR}$  can be given by means of a term deduction system  $T(\text{BPA}_{(\delta)\varepsilon} + \text{PR})$ . Its signature is the signature of  $\text{BPA}_{(\delta)\varepsilon} + \text{PR}$  and its rules are the rules of tables 12 and 14. We can define the quotient algebra of closed  $\text{BPA}_{(\delta)\varepsilon} + \text{PR}$  terms with respect to bisimulation equivalence and the operators of  $\text{BPA}_{(\delta)\varepsilon} + \text{PR}$  as usual. The quotient is a model of  $\text{BPA}_{(\delta)\varepsilon} + \text{PR}$  and it satisfies AIP. The theory  $\text{BPA}_{(\delta)\varepsilon} + \text{PR}$  is complete.

*Recursion and projection* Here we discuss the combination of recursion, projection, and the empty process. The theory  $\text{BPA}_{(\delta)\varepsilon}\text{rec} + \text{PR}$  has as its signature the signature of  $\text{BPA}_{(\delta)}\text{rec} + \text{PR}$  and a constant  $\varepsilon \notin A$  (and  $\varepsilon \neq \delta$ ). The axioms of  $\text{BPA}_{(\delta)\varepsilon}\text{rec} + \text{PR}$  are the ones of  $\text{BPA}_{(\delta)}\text{rec} + \text{PR}$  and the axioms of table 11. Moreover, we assume for axiom PR1 (table 7) that  $a$  may also be  $\varepsilon$ .

The standard facts (and their proofs) of subsection 2.4.2 are easily translated to the present situation.

The semantics of  $\text{BPA}_{(\delta)\varepsilon}\text{rec} + \text{PR}$  is given by means of a term deduction system  $T(\text{BPA}_{(\delta)\varepsilon}\text{rec} + \text{PR})$ . Its signature is the signature of  $\text{BPA}_{(\delta)\varepsilon}\text{rec} + \text{PR}$  and its rules are those of  $T(\text{BPA}_{(\delta)\varepsilon} + \text{PR})$  plus the rules concerning recursion that are presented in table 13. Since bisimulation equivalence is a congruence, we can define the operators of  $\text{BPA}_{(\delta)\varepsilon}\text{rec} + \text{PR}$  on the quotient algebra of closed  $\text{BPA}_{(\delta)\varepsilon}\text{rec} + \text{PR}$  terms with respect to bisimulation equivalence. This quotient is a model of  $\text{BPA}_{(\delta)\varepsilon}\text{rec} + \text{PR}$  and it satisfies RDP, RSP, and AIP<sup>-</sup>, but not its unrestricted version AIP. As a consequence, we can prove that  $\text{BPA}_{(\delta)\varepsilon}\text{rec}$  satisfies RSP. This is proved in the same way as theorem 2.4.36.

### 2.6.3 CCS termination

A variant of the empty process is given by the CCS process NIL [Milner, 1980]. We can extend the signature of BPA by the constant NIL, and formulate the operational rules in table 15. These rules are taken from [Aceto and Hennessy, 1992].

The crucial difference between the *necessary termination* predicate  $\checkmark$  and the termination option predicate  $\downarrow$  is in the rule for  $+$ : for  $\checkmark$ , both components must terminate in order for the sum to terminate. As a result, NIL satisfies the laws for  $\varepsilon$  but at the same time the law  $x + \text{NIL} = x$ . A

Table 15. Derivation rules of  $T(\text{BPA}_{\text{NIL}})$ .

$a \xrightarrow{a} \text{NIL} \quad \text{NIL} \checkmark$		
$\frac{x \xrightarrow{a} x'}{x + y \xrightarrow{a} x'}$	$\frac{y \xrightarrow{a} y'}{x + y \xrightarrow{a} y'}$	$\frac{x \checkmark, y \checkmark}{(x + y) \checkmark}$
$\frac{x \xrightarrow{a} x'}{xy \xrightarrow{a} x'y}$	$\frac{x \checkmark, y \xrightarrow{a} y'}{xy \xrightarrow{a} y'}$	$\frac{x \checkmark, y \checkmark}{(xy) \checkmark}$

consequence is that the law A4 (distributivity of  $\cdot$  over  $+$ ) does not hold for all processes, and so  $\text{BPA}_{\text{NIL}}$  cannot be axiomatized using the axioms of BPA. The following complete axiomatization is taken from [Baeten and Vaandrager, 1992]; for more information, we refer to this paper.

As before, we can add  $\delta$  without any operational rules. Its axioms have to be adapted in the presence of NIL, though. We show this in table 17.

## 2.7 Renaming in BPA

Sometimes it is useful to have the possibility of *renaming* atomic actions. The material of this subsection is based on [Baeten and Bergstra, 1988a] with improvements by [Vaandrager, 1990a]. Renaming operators occur in most concurrency theories; see, for example, [Milner, 1980], [Milner, 1989], [Hennessy, 1988], and [Hoare, 1985].

The signature of the equational specification  $\text{BPA} + \text{RN}$  consists of the signature of BPA plus for each function  $f$  from the set of atomic actions to itself a unary operator  $\rho_f$  called a *renaming* operator. Such a function  $f$  is called a renaming function. The axioms of  $\text{BPA} + \text{RN}$  are the ones for BPA plus the axioms concerning renaming operators displayed in table 18.

Table 16.  $\text{BPA}_{\text{NIL}}$ .

$x + y = y + x$	A1
$(x + y) + z = x + (y + z)$	A2
$x + x = x$	A3
$(ax + by + z)w = axw + (by + z)w$	A4*
$(xy)z = x(yz)$	A5
$x + \text{NIL} = x$	A6*
$x \cdot \text{NIL} = x$	A8*
$\text{NIL} \cdot x = x$	A9*



**Table 17.**  $\delta$  in the presence of NIL.

$ax + \delta = ax$	A6**
$\delta \cdot x = \delta$	A7

*Structural induction* In BPA + RN we can use structural induction just as in BPA, since closed BPA + RN terms can be rewritten into basic BPA terms. To that end, we will first prove that the term rewriting system associated to BPA + RN is strongly normalizing. We display the rewrite rules concerning the renaming operators in table 19. Note that, in rule RRN1,  $f(a)$  stands for the atomic action that  $a$  is renamed into.

**Theorem 2.7.1.** *The term rewriting system associated to BPA + RN is strongly normalizing. The rewrite rules are those of table 2 and the rules in table 19.*

**Proof.** We will apply theorem 2.2.17 to prove that the rewrite rules are terminating. For that, we first give a partial ordering of the signature.

$$\forall f, a \in A : \rho_f > \cdot > +, \rho_f > a.$$

Moreover, we give sequential composition the lexicographical status for the first argument. Now straightforward calculations will show that each left-hand side of a rewrite rule is strictly greater in the  $>_{lpo}$  ordering than its right-hand side. ■

With the aid of the above termination result, we can show the elimination theorem for basic process algebra with renaming operators.

**Theorem 2.7.2.** *For every closed BPA + RN term  $t$  there is a basic BPA term  $s$  such that  $\text{BPA} + \text{RN} \vdash t = s$ .*

**Proof.** Consider the term rewriting system presented in table 19. According to theorem 2.7.1, this term rewriting system is strongly normalizing. Now let  $t$  be a closed BPA + RN term and rewrite this into a normal form  $s$  with respect to the term rewriting system of table 19. With proposition 2.2.5 it suffices to show that  $s$  is a closed BPA term. Suppose that  $s$  contains a renaming operator and consider the smallest subterm containing

**Table 18.** Renaming.

$\rho_f(a) = f(a)$	RN1
$\rho_f(x + y) = \rho_f(x) + \rho_f(y)$	RN2
$\rho_f(xy) = \rho_f(x)\rho_f(y)$	RN3



this occurrence. The subterm has the form  $\rho_f(u)$  with  $u$  a closed BPA term. This contradicts the normality of  $s$ , since now we can rewrite the subterm using RRN1, RRN2, or RRN3. So  $s$  is a closed BPA term. ■

**Proposition 2.7.3.** *Let  $f, g : A \longrightarrow A$ . Let  $x$  be a closed BPA + RN term. Then we have the following:*

- (i)  $\rho_I(x) = x$
- (ii)  $\rho_f(\rho_g(x)) = \rho_{f \circ g}(x)$ .

Here, the function  $f \circ g : A \longrightarrow A$  is defined by  $f \circ g(a) = f(g(a))$  and the function  $I : A \longrightarrow A$  is defined by  $I(a) = a$  for all  $a \in A$ .

**Proof.** With the aid of theorem 2.7.2 it suffices to prove the theorem for basic BPA terms. For these terms the proof is trivial. ■

*Semantics* We give the semantics for BPA + RN by means of a term deduction system  $T(\text{BPA} + \text{RN})$ , whose signature is the one of BPA + RN and whose rules are the rules of tables 5 and 20. Bisimulation equivalence is a congruence, so the quotient of closed BPA + RN terms modulo bisimulation equivalence is well-defined. This means that the operators of BPA + RN can be defined on this quotient, which is a model of BPA + RN.

**Theorem 2.7.4.** *The set of closed BPA + RN terms modulo bisimulation equivalence is a model of BPA + RN.*

**Proof.** Axioms A1–A5 are treated as in 2.2.33. For RN1 take the relation that only relates  $\rho_f(a)$  and  $f(a)$ . RN2 goes like A1. RN3 goes like A5. ■

At this point we have all the ingredients necessary to state and prove that BPA + RN is a conservative extension of BPA.

**Theorem 2.7.5.** *The equational specification BPA + RN is a conservative extension of the equational specification BPA. That is, if  $t$  and  $s$  are closed BPA terms, then we have*

Table 19. A term rewriting system for BPA + RN.

$\rho_f(a) \rightarrow f(a)$	RRN1
$\rho_f(x + y) \rightarrow \rho_f(x) + \rho_f(y)$	RRN2
$\rho_f(xy) \rightarrow \rho_f(x)\rho_f(y)$	RRN3

Table 20. Derivation rules concerning renaming operators.

$\frac{x \xrightarrow{a} \sqrt{\quad}}{\rho_f(x) \xrightarrow{f(a)} \sqrt{\quad}}$	$\frac{x \xrightarrow{a} x'}{\rho_f(x) \xrightarrow{f(a)} \rho_f(x')}$
------------------------------------------------------------------------------------	------------------------------------------------------------------------

$$\text{BPA} \vdash t = s \iff \text{BPA} + \text{RN} \vdash t = s.$$

**Proof.** The operational semantics of BPA can be operationally conservatively added to the operational rules concerning the renaming operator. This follows immediately from theorem 2.4.15. The sum of these operational rules is precisely the operational semantics of BPA + RN. Now with theorem 2.4.19 we find that modulo strong bisimulation equivalence BPA + RN is an operationally conservative extension of BPA. So with theorem 2.4.24 we find with the soundness of the equational specification BPA + RN and the soundness and completeness of BPA that basic process algebra with renamings is an equationally conservative extension of BPA. ■

With the aid of the above conservativity result and the elimination theorem for BPA + RN, we find the completeness of BPA + RN.

**Theorem 2.7.6.** *The axiom system BPA + RN is a complete axiomatization of the set of closed BPA + RN terms modulo bisimulation equivalence.*

**Proof.** Apply theorem 2.4.26. ■

### 2.7.1 Extensions of BPA + RN

In this subsection we will discuss the extensions of BPA + RN with recursion and/or projections.

*Recursion* We can add recursion to BPA + RN in the same way as we added recursion to BPA. The equational specification BPAREC + RN has as its signature the signature of BPAREC plus for all functions  $f$  from  $A$  to  $A$  a renaming operator  $\rho_f$ . The equations of BPAREC + RN are the axioms of BPAREC plus the axioms concerning renaming; see table 18. We can turn the set of closed BPAREC + RN terms into a model of BPAREC + RN that satisfies RDP and RSP as usual.

*Projection* Projections can be added in an obvious way to BPA + RN: just add the projection functions and their axioms to the equational specification BPA + RN to obtain BPA + RN + PR. The standard facts that hold for BPA + PR also hold for BPA + RN + PR. As we did for BPA + PR we can infer that BPA + RN + PR is sound and complete, and that AIP<sup>-</sup> is valid.

*Recursion and projection* The extension with both recursion and projection of BPA + RN, called BPAREC + RN + PR, can be obtained just like in the case of BPA.

### 2.7.2 Renaming in basic process algebra with deadlock

In this subsection we will extend the BPA<sub>δ</sub> family with renaming operators. We begin with BPA<sub>δ</sub> itself. The equational specification BPA<sub>δ</sub> + RN has as

its signature the one of  $\text{BPA}_\delta$  and for each function  $f : A \cup \{\delta\} \rightarrow A \cup \{\delta\}$ , with  $f(\delta) = \delta$ , a unary operator  $\rho_f$  called a renaming operator. The axioms of  $\text{BPA}_\delta + \text{RN}$  are the axioms of  $\text{BPA} + \text{RN}$  plus the axioms concerning deadlock; see table 10. We assume for axiom RN1 (table 18) that  $a$  ranges over  $A \cup \{\delta\}$ . Note that we have  $\rho_f(\delta) = f(\delta) = \delta$ , for all renaming operators. This is necessary: it is easy to derive a contradiction if we allow  $\delta$  to be renamed into an atomic action.

*Structural induction* We can use structural induction as before, since closed  $\text{BPA}_\delta + \text{RN}$  terms can be rewritten into basic  $\text{BPA}_\delta$  terms. This follows from the next elimination theorem.

**Theorem 2.7.7.** *For every closed  $\text{BPA}_\delta + \text{RN}$  term  $t$  there is a basic  $\text{BPA}_\delta$  term  $s$  such that  $\text{BPA}_\delta + \text{RN} \vdash t = s$ .*

*If  $t$  and  $s$  are closed  $\text{BPA}_\delta$  terms, then we have*

$$\text{BPA}_\delta \vdash t = s \iff \text{BPA}_\delta + \text{RN} \vdash t = s.$$

**Proof.** Add the extra rewrite rule  $f(\delta) \rightarrow \delta$  to table 19 and reiterate the proof of theorem 2.7.2. ■

**Remark 2.7.8.** Note that proposition 2.7.3 also holds for  $\text{BPA}_\delta + \text{RN}$ .

*Semantics* The semantics for  $\text{BPA}_\delta + \text{RN}$  can be given just like the semantics for  $\text{BPA} + \text{RN}$ . Let  $T(\text{BPA}_\delta + \text{RN})$  be the term deduction system with the signature of  $\text{BPA}_\delta + \text{RN}$  and with the rules of tables 5 and 20. In the latter table we further assume that  $f(a) \in A$ . Since bisimulation equivalence is a congruence, the quotient of the set of closed  $\text{BPA}_\delta + \text{RN}$  terms with respect to bisimulation equivalence is well-defined. This quotient is a model of  $\text{BPA}_\delta + \text{RN}$ ; from this, the completeness of  $\text{BPA}_\delta$ , and the elimination result, the completeness of  $\text{BPA}_\delta + \text{RN}$  follows.

### 2.7.3 Extensions of $\text{BPA}_\delta + \text{RN}$ and $\text{BPA}_\delta$

In this subsection we discuss the extensions of  $\text{BPA}_\delta + \text{RN}$  with recursion and/or projections and we discuss the extension of  $\text{BPA}_\delta$  with a particular renaming operator.

*Recursion and/or projection* The extensions of  $\text{BPA}_\delta + \text{RN}$  with recursion, projection, or a combination of both are obtained in the same way as these extensions without deadlock; see section 2.7.1.

*Encapsulation* In most concurrency theories in which a form of deadlock is present there is usually the notion of an *encapsulation* or *restriction* operator; this is a renaming operator that renames certain atomic actions into  $\delta$ . The notion of encapsulation and the notation  $\partial_H$  stem from [Bergstra and Klop, 1984b]. The notion named restriction is due to [Milner, 1980].

In this subsection we will add the encapsulation operator to  $\text{BPA}_\delta$ .

**Table 21.** The encapsulation operator.

$\partial_H(a) = a, \text{ if } a \notin H$	D1
$\partial_H(a) = \delta, \text{ if } a \in H$	D2
$\partial_H(x + y) = \partial_H(x) + \partial_H(y)$	D3
$\partial_H(xy) = \partial_H(x)\partial_H(y)$	D4

The equational specification  $\text{BPA}_\delta + \partial_H$  has as its signature the one of  $\text{BPA}_\delta$  plus for each  $H \subseteq A$  a unary operator  $\partial_H$  called the encapsulation operator. The axioms of  $\text{BPA}_\delta + \partial_H$  are the equations of  $\text{BPA}_\delta$  and the equations defining  $\partial_H$  in table 21. We assume in this table that  $a$  ranges over  $A \cup \{\delta\}$ , so in particular we find with D1 that  $\partial_H(\delta) = \delta$ .

The semantics of  $\text{BPA}_\delta + \partial_H$  can be derived just like in the case of  $\text{BPA}_\delta + \text{RN}$ . We can take  $\partial_H = \rho_{f_H}$  with

$$f_H(a) = \begin{cases} a & \text{if } a \notin H; \\ \delta & \text{otherwise.} \end{cases}$$

For completeness sake, we give the operational rules for the encapsulation operator in table 22.

It is also straightforward to extend  $\text{BPA}_\delta + \partial_H$  with recursion and/or projections.

#### 2.7.4 Renaming in basic process algebra with empty process

In this subsection we will add renaming operators to both  $\text{BPA}_\varepsilon$  and  $\text{BPA}_{\delta\varepsilon}$  with extensions. We will simultaneously refer to both of them as before with parentheses:  $\text{BPA}_{(\delta)\varepsilon}$ . Arbitrary combinations of renaming operators and the empty process introduce a form of abstraction, which is beyond the scope of concrete process algebra. Therefore, we will restrict ourselves to the concrete subcase that prohibits renaming into the empty process.

The signature of the equational specification  $\text{BPA}_{(\delta)\varepsilon} + \text{RN}$  is the signature of  $\text{BPA}_{(\delta)} + \text{RN}$  plus a constant  $\varepsilon \notin A$  ( $\varepsilon \neq \delta$ ). We do not allow renaming into  $\varepsilon$  so for the functions  $f$  we assume (moreover) that  $f(a) \in A_{(\delta)}$  if  $a \in A_{(\delta)}$  and  $(f(\delta) = \delta, f(\varepsilon) = \varepsilon)$ . The axioms of  $\text{BPA}_{(\delta)\varepsilon} + \text{RN}$  are the ones of  $\text{BPA}_{(\delta)\varepsilon}$  and the equations for renaming; see table 18. Note that  $\rho_f(\varepsilon) = \varepsilon$  (and  $\rho_f(\delta) = \delta$ ).

**Table 22.** Derivation rules for the encapsulation operator.

$\frac{x \xrightarrow{a} \sqrt{\quad}}{\partial_H(x) \xrightarrow{a} \sqrt{\quad}}, a \notin H$	$\frac{x \xrightarrow{a} x'}{\partial_H(x) \xrightarrow{a} \partial_H(x')}, a \notin H$
-------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------



*Abstraction* We have an abstraction mechanism if we allow renaming into the empty process. For instance, suppose that we have two atomic actions, say  $a$  and  $b$ . Let  $f(a) = a$  and  $f(b) = \varepsilon$ . Then  $\rho_f(ab) = a$  and we have abstracted from  $b$ .

*Structural induction* We can use structural induction as before, since closed  $\text{BPA}_{(\delta)\varepsilon} + \text{RN}$  terms can be rewritten into basic  $\text{BPA}_{(\delta)\varepsilon}$  terms. This can be shown along the same lines as the elimination theorem for the theory without the empty process; see, for instance, theorem 2.7.2.

Note that proposition 2.7.3 also holds for  $\text{BPA}_{(\delta)\varepsilon} + \text{RN}$ .

*Semantics* The semantics of  $\text{BPA}_{(\delta)\varepsilon} + \text{RN}$  will be given by means of a term deduction system. Let  $T(\text{BPA}_{(\delta)\varepsilon} + \text{RN})$  be the term deduction system with  $\text{BPA}_{(\delta)\varepsilon} + \text{RN}$  as its signature and with rules displayed in tables 12 and 23. Bisimulation equivalence is a congruence, so we can define the operators of  $\text{BPA}_{(\delta)\varepsilon} + \text{RN}$  on the quotient of closed  $\text{BPA}_{(\delta)\varepsilon} + \text{RN}$  terms modulo bisimulation equivalence. It is straightforward to prove that this is a model of  $\text{BPA}_{(\delta)\varepsilon} + \text{RN}$ . The completeness of  $\text{BPA}_{(\delta)\varepsilon} + \text{RN}$  is also standard to prove.

*Look-ahead* If we allow renaming into the empty process, we need two more derivation rules that concern renaming; they introduce a look-ahead as can be seen in table 24. The operational rules that we list in this table are due to [Baeten and Glabbeek, 1987].

We will give an example. Suppose that  $f(a) = \varepsilon$  and  $f(b) = b$ . Then we can derive  $\rho_f(a^n b) \xrightarrow{b} \varepsilon$ , for each  $n \geq 1$ .

### 2.7.5 Extensions of $\text{BPA}_{(\delta)\varepsilon} + \text{RN}$ and $\text{BPA}_{(\delta)\varepsilon}$

In this subsection we discuss the extensions of  $\text{BPA}_{(\delta)\varepsilon} + \text{RN}$  with recursion and/or projections and we discuss the extension of  $\text{BPA}_{(\delta)\varepsilon}$  with a particular renaming operator.

*Recursion and/or projection* The extensions of  $\text{BPA}_{(\delta)\varepsilon} + \text{RN}$  with recursion and/or projection can be obtained just like before. However, if we

**Table 23.** Derivation rules for renaming operators and empty process.

$\frac{x \downarrow}{\rho_f(x) \downarrow}$	$\frac{x \xrightarrow{a} x'}{\rho_f(x) \xrightarrow{f(a)} \rho_f(x')}, f(a) \in A$
---------------------------------------------	------------------------------------------------------------------------------------

**Table 24.** Extra rules when we allow renaming into  $\varepsilon$ .

$\frac{x \xrightarrow{a} y, \rho_f(y) \downarrow}{\rho_f(x) \downarrow}, f(a) = \varepsilon$	$\frac{x \xrightarrow{a} y, \rho_f(y) \xrightarrow{b} x'}{\rho_f(x) \xrightarrow{b} x'}, f(a) = \varepsilon$
----------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------



allow renaming into the empty process the definition of a guarded recursive specification has to be adapted. We will show in an example that RSP no longer holds with the present definition. This example is taken from [Baeten *et al.*, 1987]. Suppose that we have at least three elements in the set of atomic actions, say  $a, i$ , and  $j$ . Let  $\varepsilon_i$  resp.  $\varepsilon_j$  be the renaming operators that rename  $i$  resp.  $j$  into  $\varepsilon$  and further do nothing. Then the guarded recursive specification

$$\{X = i \cdot \varepsilon_j(Y), Y = j \cdot \varepsilon_i(X)\}$$

has the solution  $\{ia^n, ja^n\}$  for all  $n \geq 1$ . So RSP cannot hold.

A possible solution can be to prohibit the occurrences of renaming operators in the body of guarded recursive specifications. Also more sophisticated solutions can be obtained in terms of restrictions on the renaming operators that do occur in the body of a guarded recursive specification.

*Encapsulation* The extension of  $\text{BPA}_{\delta\varepsilon}$  with the encapsulation operator can be obtained in the same way as in the case without the empty process; see subsection 2.7.3.

## 2.8 The state operator

In this subsection we extend BPA with the (simple) state operator, which is a generalization of a renaming operator. It is a renaming operator with a memory to describe processes with an independent global state. We denote the state operator by  $\lambda_s$ ; the subscript is the memory cell containing the current state  $s$ . This subsection is based on [Baeten and Bergstra, 1988a]. Another treatment of state operators can be found in [Verhoef, 1992].

Next, we will discuss the signature of  $\text{BPA}_\lambda$ . It consists of the usual signature of BPA extended for each  $m \in M$  and  $s \in S$  with a unary operator  $\lambda_s^m$  called the (simple) state operator.  $M$ ,  $S$ , and  $A$  are mutually disjoint. The set  $S$  is the state space and  $M$  is the set of object names; the  $M$  stands for machine.

We describe the state operator by means of two total functions *action* and *effect*. The function *action* describes the renaming of the atomic actions and the function *effect* describes the contents of the memory. We have

$$\text{action} : A \times M \times S \longrightarrow A, \quad \text{effect} : A \times M \times S \longrightarrow S.$$

Mostly, we write  $a(m, s)$  for  $\text{action}(a, m, s)$  and  $s(m, a)$  for  $\text{effect}(a, m, s)$ .

Intuitively, we think of the process  $\lambda_s^m(x)$  as follows:  $m$  represents a machine (say a computer),  $s$  describes its state (say the contents of its memory),  $x$  is its input (say a program). Now  $\lambda_s^m(x)$  describes what happens when  $x$  is presented to machine  $m$  in state  $s$ .

Now we discuss the equations of  $\text{BPA}_\lambda$ . They are the axioms of BPA

**Table 25.** The axioms defining the state operator.

$\lambda_s^m(a) = a(m, s)$	SO1
$\lambda_s^m(ax) = a(m, s)\lambda_{s(m,a)}^m(x)$	SO2
$\lambda_s^m(x + y) = \lambda_s^m(x) + \lambda_s^m(y)$	SO3

(see table 1) and the axioms of table 25. The first axiom SO1 gives the renaming part of the state operator. The second axiom SO2 shows the effect of renaming an atomic action on the current state. Note that if a renaming has no effect on states we obtain an ordinary renaming operator. Axiom SO3 expresses that the state operator distributes over the alternative composition.

### 2.8.1 Termination and elimination

Next, it is our aim to show that the state operator can be eliminated. Therefore, we will use that the term rewriting system associated to  $\text{BPA}_\lambda$  is strongly normalizing. We will prove the latter fact with the aid of the method of the recursive path ordering. However, we cannot apply this method immediately. This is due to the fact that we cannot hope to find a strict partial ordering on the signature of  $\text{BPA}_\lambda$  that does the job. The problematical rule is the rewrite rule RSO2 (see table 26). Suppose that we have one atomic action  $a$ . Let us have two different states, which shall remain nameless. Take an inert action function, that is it does nothing, and let the effect function act as a switch. This yields the following instantiation for the rewrite rule RSO2:

$$\begin{aligned}\lambda(ax) &\rightarrow a\lambda'(x), \\ \lambda'(ax) &\rightarrow a\lambda(x).\end{aligned}$$

For the first rewrite rule the ordering that works is  $\lambda > \lambda'$ . But for the second rule, the ordering should be the opposite, thus yielding an inconsistency. We solve this by giving the state operator a rank; the rank of a state operator depends on the weight of its operand. This idea is taken from [Verhoef, 1992]; he based this idea on a method that [Bergstra and Klop, 1985] give for the termination of a concurrent system (see theorem 3.2.3 where we treat their method).

**Definition 2.8.1.** Let  $x$  and  $y$  be terms and let  $a$  be an atomic action. The weight of a term  $x$ , notation  $|x|$  is defined inductively as follows:

- $|a| = 1$ ,
- $|x + y| = \max\{|x|, |y|\}$ ,
- $|x \cdot y| = |x| + |y|$ ,

$$\bullet |\lambda_s^m(x)| = |x|.$$

**Definition 2.8.2.** The rank of a state operator is the weight of the sub-term of which it is the leading operator. So, if  $|x| = n$ , we write  $\lambda_{n,s}^m(x)$ .

**Theorem 2.8.3.** *The term rewriting system associated to the equational specification of  $\text{BPA}_\lambda$  is strongly normalizing. The rewrite rules are the ones listed in tables 2 and 26.*

**Proof.** Take the following precedence for the elements of the signature of  $\text{BPA}_\lambda$ :

$$\forall n \geq 1, m \in M, s, s' \in S, a \in A : \lambda_{n+1,s}^m > \lambda_{n,s'}^m > \cdot > +, \lambda_{n,s}^m > a.$$

Moreover, give the sequential composition the lexicographical status for the first argument. Now it is not hard to see that each left-hand side of the rewrite rules is strictly greater than its right-hand side in the  $>_{lpo}$  ordering. We will treat an example. Let  $a' = a(m, s)$ ,  $\lambda_n = \lambda_{n,s}^m$ , and  $\lambda'_n = \lambda_{n,s(m,a)}^m$ . Suppose that  $|x| = n$ .

$$\begin{aligned} \lambda_{n+1}(ax) &>_{lpo} \lambda_{n+1}^*(ax) \\ &>_{lpo} \lambda_{n+1}^*(ax) \cdot \lambda_{n+1}^*(ax) \\ &>_{lpo} a' \cdot \lambda'_n(\lambda_{n+1}^*(ax)) \\ &>_{lpo} a' \cdot \lambda'_n(ax) \\ &>_{lpo} a' \cdot \lambda_n^*(ax) \\ &>_{lpo} a' \cdot \lambda'_n(a \cdot^* x) \\ &>_{lpo} a' \cdot \lambda'_n(x). \end{aligned}$$

The other inequalities are checked analogously. With theorem 2.2.17 it follows that the system is terminating. ■

Now, we can state the elimination theorem for basic process algebra with the state operator.

**Theorem 2.8.4.** *For every closed  $\text{BPA}_\lambda$  term  $t$  there is a basic BPA term  $s$  such that  $\text{BPA}_\lambda \vdash t = s$ .*

**Proof.** Straightforward. ■

**Table 26.** The rewrite rules for the simple state operator.

$\lambda_s^m(a) \rightarrow a(m, s)$	RSO1
$\lambda_s^m(ax) \rightarrow a(m, s)\lambda_{s(m,a)}^m(x)$	RSO2
$\lambda_s^m(x + y) \rightarrow \lambda_s^m(x) + \lambda_s^m(y)$	RSO3

**Table 27.** Derivation rules of  $T(\text{BPA}_\lambda)$ .

$\frac{x \xrightarrow{a} \surd}{\lambda_s^m(x) \xrightarrow{a(m,s)} \surd}$	$\frac{x \xrightarrow{a} x'}{\lambda_s^m(x) \xrightarrow{a(m,s)} \lambda_{s(m,a)}^m(x')}$
-----------------------------------------------------------------------------	-------------------------------------------------------------------------------------------

*Semantics* We give the semantics for  $\text{BPA}_\lambda$  by means of a term deduction system  $T(\text{BPA}_\lambda)$ . Its signature is that of  $\text{BPA}_\lambda$  and its rules are the rules of tables 5 and 27. According to theorem 2.2.31 bisimulation equivalence is a congruence so the operators of  $\text{BPA}_\lambda$  can be defined on the quotient of the closed  $\text{BPA}_\lambda$  terms modulo bisimulation equivalence. Moreover, it is a model of  $\text{BPA}_\lambda$ . With the aid of the method explained in subsection 2.4.1, it is not hard to see that  $\text{BPA}_\lambda$  is a conservative extension of  $\text{BPA}$ . Then it easily follows with theorem 2.4.26 that the axioms in tables 1 and 25 constitute a complete axiomatization of  $\text{BPA}_\lambda$ .

*Extensions* The extensions of  $\text{BPA}_\lambda$  with recursion and/or projection are obtained in the same way as those of  $\text{BPA}$ .

The extension of  $\text{BPA}_\delta$  with the state operator is obtained in the same way as the extension of  $\text{BPA}$  with it. We also allow  $a(m, s) = \delta$  so the action function can rename into  $\delta$ . There is only one extra axiom: we need to know what the state operator should do with the extra constant  $\delta$ . Therefore, we need to know how the functions *action* and *effect* are extended to  $A \cup \{\delta\}$ . We define  $\delta(m, s) = \delta$  and  $s(m, \delta) = s$ . The extra axiom is

$$\lambda_s^m(\delta) = \delta.$$

The extensions of  $\text{BPA}_{\delta\lambda}$  with recursion and/or projection are obtained in the same way as those of  $\text{BPA}_\delta$ .

The following example is due to Alban Ponse [Ponse, 1993].

**Example 2.8.5.** We describe an edit session with the aid of the state operator. We will use the theory  $\text{BPA}_\lambda$  with recursion.

The characters that can be typed are the lower case characters  $a, b, \dots, z$  with the usual meaning, and two special characters  $D$  and  $P$ . We call the set of characters that can be typed  $C$ . The character  $D$  stands for the deletion of the last character from the memory; if the memory is empty pressing the  $D$  will cause a *beep*. The  $P$  sends the contents of the memory to a printer device. We have a user  $U$  that wants to type characters from  $C$ . The user is specified as follows:

$$U = \sum_{c \in C} \text{type}(c) \cdot U + \sum_{c \in C} \text{type}(c).$$

The state space is  $S = \{a, b, \dots, z\}^*$ ; we denote the empty word by  $\varepsilon$ . The

set  $A$  of atomic actions is

$$\{type(c), typed(c), deleted(c) : c \in C\} \cup \{printed(\sigma) : \sigma \in S\} \cup \{beep\}.$$

We give the *action* and *effect* functions implicitly, by giving the relevant axioms for our specific state operator. We assume that  $c \in \{a, b, \dots, z\}$  and  $\sigma \in S$ .

$$\begin{aligned} \lambda_\epsilon(type(D) \cdot x) &= beep \cdot \lambda_\epsilon(x) \\ \lambda_{\sigma c}(type(D) \cdot x) &= deleted(c) \cdot \lambda_\sigma(x) \\ \lambda_\sigma(type(c) \cdot x) &= typed(c) \cdot \lambda_{\sigma c}(x) \\ \lambda_\sigma(type(P) \cdot x) &= printed(\sigma) \cdot \lambda_\sigma(x). \end{aligned}$$

For the other atomic actions in  $A$  we define the *action* and *effect* functions to be inert. Now the process  $\lambda_\epsilon(U)$  describes an edit session. Since we have only one object name, we left out the superscripts.

Note that the following choice for the last equation of the above display also works:

$$\lambda_\sigma(type(P) \cdot x) = printed(\sigma) \cdot \lambda_\epsilon(x).$$

However, we did not choose this option to separate different concerns: if we want to empty the memory, it may be more appropriate to define an atomic action that empties the memory.

## 2.9 The extended state operator

In the following, we will discuss BPA with the extended state operator, which is a generalization of the simple state operator. We denote the extended state operator by  $\Lambda_s^m$ . The difference with the (simple) state operator is that we can rename an atomic action into a closed term of a particular form, namely a finite sum of atomic actions. With this extra feature it is possible to translate an instruction like `read(x)` into process algebra.

This subsection is based on [Baeten and Bergstra, 1988a].

We discuss the signature of  $BPA_\Lambda$ . It consists of the usual signature of BPA extended for each  $m \in M$  and  $s \in S$  with a unary operator  $\Lambda_s^m$  called the extended state operator.  $M$ ,  $S$ , and  $A$  are mutually disjoint. The set  $S$  is the state space and  $M$  is the set of object names; the  $M$  stands for machine.

We describe the extended state operator by means of two functions *action* and *effect*. The function *action* describes the renaming of the atomic actions and the function *effect* describes the contents of the memory. We have

$$action : A \times M \times S \longrightarrow 2^A \setminus \{\emptyset\}, \quad effect : A \times M \times S \times A \longrightarrow S.$$



**Table 28.** The axioms defining the generalized state operator.

$\Lambda_s^m(a) = \sum_{b \in a(m,s)} b$	GS1
$\Lambda_s^m(ax) = \sum_{b \in a(m,s)} b \cdot \Lambda_{s(m,a,b)}^m(x)$	GS2
$\Lambda_s^m(x + y) = \Lambda_s^m(x) + \Lambda_s^m(y)$	GS3

**Table 29.** Derivation rules of  $T(\text{BPA}_\Lambda)$ .

$\frac{x \xrightarrow{a} \sqrt{\quad}}{\Lambda_s^m(x) \xrightarrow{b} \sqrt{\quad}}, b \in a(m,s)$	$\frac{x \xrightarrow{a} x'}{\Lambda_s^m(x) \xrightarrow{b} \Lambda_{s(m,a,b)}^m(x')}, b \in a(m,s)$
----------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------

We write  $a(m, s)$  for *action*( $a, m, s$ ) and  $s(m, a, b)$  for *effect*( $a, m, s, b$ ).

The axioms of  $\text{BPA}_\Lambda$  are those of BPA and the axioms of table 28. Next, we discuss them. The first axiom GS1 states that an atomic action is renamed into a sum of atomic actions. Axiom GS2 shows the side effects of the renaming on the state space. Axiom GS3 expresses that the extended state operator distributes over the alternative composition.

*Termination* In the previous subsection (2.8) we mentioned that we cannot use the method of the recursive path ordering immediately. The same phenomenon occurs with the extended state operator. Fortunately, the solution of the problems is the same as for the simple state operator. We have to define ranked extended state operators and prove the termination of this system. We omit the details and refer to subsection 2.8 for more information. We only mention the main result.

**Theorem 2.9.1.** *The term rewriting system that is associated to  $\text{BPA}_\Lambda$  is strongly normalizing. The rewrite rules are those of tables 2 and 30.*

*Semantics* We give the semantics for  $\text{BPA}_\Lambda$  by means of a term deduction system  $T(\text{BPA}_\Lambda)$ . Its signature is that of  $\text{BPA}_\Lambda$  and its rules are the rules of tables 5 and 29. According to 2.2.31 bisimulation equivalence is a congruence so the quotient of the closed  $\text{BPA}_\Lambda$  terms modulo bisimulation equivalence is well-defined. Moreover, it is easily seen that the quotient is a model of  $\text{BPA}_\Lambda$ . With the theory of section 2.4.1, we find that  $\text{BPA}_\Lambda$  is a conservative extension of BPA. With the termination theorem 2.9.1 and an elimination result, similar to 2.8.4, we find using theorem 2.4.26 that the axioms of tables 1 and 28 constitute a complete axiomatization of  $\text{BPA}_\Lambda$ .

*Extensions* The extensions of  $\text{BPA}_\Lambda$  and  $\text{BPA}_{\delta\Lambda}$  with recursion and/or projection are obtained in the same way as those with  $\text{BPA}_\lambda$  and  $\text{BPA}_{\delta\lambda}$ .

**Table 30.** The rewrite rules for the extended state operator.

$\Lambda_s^m(a) \rightarrow \sum_{b \in a(m,s)} b$	RGS1
$\Lambda_s^m(ax) \rightarrow \sum_{b \in a(m,s)} b \cdot \Lambda_{s(m,a,b)}^m(x)$	RGS2
$\Lambda_s^m(x + y) \rightarrow \Lambda_s^m(x) + \Lambda_s^m(y)$	RGS3

The only difference is that we can now allow  $a(m, s) = \emptyset$  if in addition we define

$$\sum_{b \in \emptyset} b = \delta.$$

As before, we have  $\Lambda_s^m(\delta) = \delta$ .

**Example 2.9.2.** In this example we describe a gambling session of a fruit machine player with the aid of the extended state operator. We use the theory  $\text{BPA}_\Lambda$  with recursion, again leaving out superscripts.

The player  $P$  is specified as follows:

$$P = \text{pull} \cdot \text{win} \cdot P.$$

Note that  $P$  has a serious gambling problem. With the extended state operator we specify what actually will happen during the gambling session. First we give the state space  $S = F \times F \times F$  where

$$F = \{\text{bar}, \text{bell}, \text{grape}, \text{melon}, \text{orange}, \text{cherry}\}.$$

The set of atomic actions  $A$  is

$$\{\text{pull}, \text{win}, \text{lost}\} \cup \{\text{won}(f) : f \in F\} \cup \{\text{pulled}(f, g, h) : f, g, h \in F\}.$$

We define the *action* and *effect* functions implicitly by giving the relevant instances of axiom GS2. The first equation expresses that if  $P$  pulls the fruit machine it will give one of the possible triples. The second equation describes that *win* is renamed into *lost* if the obtained triple contains different “fruits”. If the triple contains only one symbol, say *melon*, we have that *win* is renamed into *won(melon)*.

$$\begin{aligned} \Lambda_{(u,v,w)}(\text{pull} \cdot x) &= \sum_{(f,g,h) \in S} \text{pulled}(f, g, h) \cdot \Lambda_{(f,g,h)}(x) \\ \Lambda_{(f,g,h)}(\text{win} \cdot x) &= \sum_{f=g=h} \text{won}(f) \cdot \Lambda_{(f,g,h)}(x) + \sum_{f \neq g} \text{lost} \cdot \Lambda_{(f,g,h)}(x) \end{aligned}$$

$$+ \sum_{g \neq h} \text{lost} \cdot \Lambda_{(f,g,h)}(x) + \sum_{f \neq h} \text{lost} \cdot \Lambda_{(f,g,h)}(x).$$

For the other actions in  $A$  we define both functions *action* and *effect* to be inert. The process  $\Lambda_{(f,g,h)}(P)$  with  $f, g, h \in F$  describes a gambling session.

## 2.10 The priority operator

In this subsection we introduce  $\text{BPA}_\delta$  with the priority operator that originates from [Baeten *et al.*, 1986].

The signature of the equational specification  $\text{BPA}_\delta$  with the priority operator,  $\text{BPA}_{\delta\theta}$ , consists of the signature of  $\text{BPA}_\delta$  plus a unary operator  $\theta$  and an auxiliary binary operator  $\triangleleft$  pronounced “unless”. Furthermore, a partial ordering  $<$ , called the *priority ordering* on the set of atomic actions  $A$  is presumed. The axioms of the equational specification  $\text{BPA}_{\delta\theta}$  are the usual axioms of  $\text{BPA}_\delta$  (see tables 1 and 10) and the axioms of tables 31 and 32. The axioms that we present in these tables make use of  $\delta$ . We can imagine a system without  $\delta$  ( $\text{BPA}_\theta$ ) but such a system has a laborious axiomatization; see [Bergstra, 1985] for such an axiomatization.

Next, we will discuss the axioms concerning priority.

The axioms of table 31 define the auxiliary unless operator. It is used to axiomatize the priority operator. The intended behaviour of the unless operator is that the process  $x \triangleleft y$  filters out all summands of  $x$  with an initial action smaller than some initial action of  $y$ . So, one could say that the second argument  $y$  is the filter. If, for instance,  $a > b > c$  then we want that

$$(ax + by + cz) \triangleleft (bp + cq) = ax + by.$$

To model the filter behaviour we use the constant process  $\delta$  to rename the unwanted initial actions of  $x$  into  $\delta$ . The axioms U1 and U2 essentially define the mesh of the filter: they say which actions can pass the filter and which cannot. Axiom U3 expresses the fact that the initial actions of  $y$

Table 31. The axioms defining the unless operator.

$a \triangleleft b = a$	if $\neg(a < b)$	U1
$a \triangleleft b = \delta$	if $a < b$	U2
$x \triangleleft yz = x \triangleleft y$		U3
$x \triangleleft (y + z) = (x \triangleleft y) \triangleleft z$		U4
$xy \triangleleft z = (x \triangleleft z)y$		U5
$(x + y) \triangleleft z = x \triangleleft z + y \triangleleft z$		U6

**Table 32.** The axioms defining the priority operator.

$\theta(a) = a$	TH1
$\theta(xy) = \theta(x) \cdot \theta(y)$	TH2
$\theta(x + y) = \theta(x) \triangleleft y + \theta(y) \triangleleft x$	TH3

are the same as the initial actions of  $yz$ . Axiom U4 says that it is the same to filter the initial actions of  $x$  with filter  $y + z$  as to filter first the initial actions of  $x$  with filter  $y$  and filter the result with filter  $z$ . Axiom U5 expresses that  $z$  is a disposable filter: once in  $xy$  the process  $x$  is filtered through  $z$ , the process  $y$  can freely pass. Axiom U6 expresses that filtering a sum is the same as adding the filtered summands.

The priority operator uses the unless operator to filter out the summands with low priority. Thus, the priority operator is invariant under atomic actions and sequential composition. This is expressed in the axioms TH1 and TH2. The priority operator does *not* distribute over the alternative composition, since in a prioritized sum  $\theta(x + y)$  there is an interaction between the restrictions concerning the priorities imposed on each other by  $x$  and  $y$ , whereas in  $\theta(x) + \theta(y)$  we do not have such an interaction. Axiom TH3 states that the prioritized sum equals the sum of the prioritized summands with the remaining alternatives as filters. So, for instance, we have

$$\theta(a + b + c) = \theta(a) \triangleleft (b + c) + \theta(b) \triangleleft (a + c) + \theta(c) \triangleleft (a + b).$$

*Intuition* The partial order  $<$  is used in order to describe which actions have priority over other actions. If for instance  $a < b$  and  $b$  and  $c$  are not related we want to have that  $\theta(a + b) = b$  and  $\theta(b + c) = b + c$ . The priority operator thus respects the alternative composition for actions without priority but gives the alternative with the highest priority, in the  $<$  hierarchy, if the sum contains prioritized actions. A typical example of a low priority action is an atomic action expressing time-out behaviour: as long as there are alternatives with a higher priority no time-out will be performed within the scope of the priority operator. The priority operator has been used to specify and verify time critical protocols in an untimed setting; see, for instance, [Vaandrager, 1990b].

Next, we list some properties of the unless operator and the priority operator that can be derived from  $\text{BPA}_{\delta\theta}$ . The first identity expresses that the ordering of filtering does not matter. The second equation expresses that when a process is filtered once, a second application of the same filter

**Table 33.** Rewrite rules for the unless operator.

$\neg(a < b) \implies a \triangleleft b \rightarrow a$	RU1
$a < b \implies a \triangleleft b \rightarrow \delta$	RU2
$x \triangleleft yz \rightarrow x \triangleleft y$	RU3
$x \triangleleft (y + z) \rightarrow (x \triangleleft y) \triangleleft z$	RU4
$xy \triangleleft z \rightarrow (x \triangleleft z)y$	RU5
$(x + y) \triangleleft z \rightarrow x \triangleleft z + y \triangleleft z$	RU6
$(x \triangleleft y) \triangleleft y \rightarrow x \triangleleft y$	RU7

has no effect. The third one expresses that a prioritized process  $\theta(x)$  is automatically filtered with its subprocess  $x$  without priority.

**Lemma 2.10.1.** *The following identities are derivable from  $\text{BPA}_{\delta\theta}$ :*

- $(x \triangleleft y) \triangleleft z = (x \triangleleft z) \triangleleft y$ ,
- $(x \triangleleft y) \triangleleft y = x \triangleleft y$ ,
- $\theta(x) \triangleleft x = \theta(x)$ .

**Proof.** The proofs of these identities are easy. To illustrate the usage of the axioms we provide full proofs. Here is the first one:

$$(x \triangleleft y) \triangleleft z = x \triangleleft (y + z) = x \triangleleft (z + y) = (x \triangleleft z) \triangleleft y.$$

For the second one, take  $z = y$  in the above deduction and use the fact that  $y + y = y$ . The third identity is derived as follows:

$$\theta(x) \triangleleft x = \theta(x) \triangleleft x + \theta(x) \triangleleft x = \theta(x + x) = \theta(x).$$

Note the double use of the idempotency of the alternative composition in this inference. ■

Next, we formulate a term rewriting result for basic process algebra with priorities. It states that the term rewriting system associated to  $\text{BPA}_{\delta\theta}$  is strongly normalizing. To prove this we use the method of the recursive path ordering that we introduced in subsection 2.2.2. We need the lexicographical variant of this method. Note that the rewrite rules concerning the unless operator (table 33) form a *conditional* term rewriting system. We can, however, see the rewrite rules RU1 and RU2 as a scheme of rules; for all  $a$  and  $b$  there is a rule. So, in fact, this term rewriting system is unconditional. Thus, we may use the method of the recursive path ordering.

**Theorem 2.10.2.** *The term rewriting system that is associated to  $\text{BPA}_{\delta\theta}$  is strongly normalizing. This term rewriting system consists of the rewrite rules listed in table 2, table 33, and table 34.*



**Table 34.** The rewrite rules for the priority operator.

$\theta(a) \rightarrow a$	RTH1
$\theta(xy) \rightarrow \theta(x) \cdot \theta(y)$	RTH2
$\theta(x + y) \rightarrow \theta(x) \triangleleft y + \theta(y) \triangleleft x$	RTH3
$\theta(x) \triangleleft x \rightarrow \theta(x)$	RTH4

**Proof.** We use the lexicographical variant of the recursive path ordering that we treated in subsection 2.2.2. Take as precedence for the elements of the signature of  $\text{BPA}_{\delta\theta}$  the following partial ordering:

$$\theta > \triangleleft > \cdot > +, \quad \forall a \in A : a > \delta.$$

Furthermore, we give the sequential composition the lexicographical status for the first argument and we give the unless operator the lexicographical status for the second argument. We will treat a typical case: we treat the case RU4 where we use the lexicographical status of the unless operator.

$$\begin{aligned}
 x \triangleleft (y + z) &>_{lpo} x \triangleleft^* (y + z) \\
 &>_{lpo} (x \triangleleft^* (y + z)) \triangleleft (y +^* z) \\
 &>_{lpo} (x \triangleleft (y +^* z)) \triangleleft z \\
 &>_{lpo} (x \triangleleft y) \triangleleft z.
 \end{aligned}$$

The other cases are dealt with in a similar way. This means that we find with theorem 2.2.17 that the term rewriting system is strongly normalizing, which ends the proof of the theorem. ■

Next, we formulate the elimination theorem for basic process algebra with priorities.

**Theorem 2.10.3.** *The equational specification  $\text{BPA}_{\delta\theta}$  has the elimination property for  $\text{BPA}_{\delta}$ . That is, for every closed  $\text{BPA}_{\delta\theta}$  term  $t$  there is a basic  $\text{BPA}_{\delta}$  term  $s$  such that  $\text{BPA}_{\delta\theta} \vdash t = s$ .*

**Proof.** Easy. ■

### 2.10.1 Semantics of basic process algebra with priorities

In this subsection we discuss the operational semantics of  $\text{BPA}_{\delta\theta}$ .

The operational semantics of the priority operator can be found in [Baeten and Bergstra, 1988b]. A more accessible reference is, for instance, [Groote, 1990b] or [Baeten and Weijland, 1990]. In table 35 we give the characterization presented in [Baeten and Weijland, 1990]. In [Bol and Groote, 1991] we find rules that operationally define the unless operator,

**Table 35.** Derivation rules for the priority operator.

$\frac{x \xrightarrow{a} x', \{x \not\xrightarrow{b}, x \not\xrightarrow{b} \sqrt{\mid} b > a\}}{\theta(x) \xrightarrow{a} \theta(x')}$	$\frac{x \xrightarrow{a} \sqrt{\mid}, \{x \not\xrightarrow{b}, x \not\xrightarrow{b} \sqrt{\mid} \mid b > a\}}{\theta(x) \xrightarrow{a} \sqrt{\mid}}$
-----------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------

**Table 36.** Derivation rules for the unless operator.

$\frac{x \xrightarrow{a} x', \{y \not\xrightarrow{b}, y \not\xrightarrow{b} \sqrt{\mid} \mid b > a\}}{x \triangleleft y \xrightarrow{a} x'}$	$\frac{x \xrightarrow{a} \sqrt{\mid}, \{y \not\xrightarrow{b}, y \not\xrightarrow{b} \sqrt{\mid} \mid b > a\}}{x \triangleleft y \xrightarrow{a} \sqrt{\mid}}$
----------------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------

essentially as in table 36 (but we follow the approach of [Baeten and Weijland, 1990]).

We note that it is possible to operationally characterize the priority operator without the use of the unless operator. The latter one is used for the axiomatization of the priority operator. However, [Bergstra, 1985] gives a not so well-known finite axiomatization of the priority operator without the unless operator. Moreover, in this approach the special constant  $\delta$  is not necessary. For more information on this axiomatization we refer to [Bergstra, 1985].

We also note that on the basis of an operational semantics for the priority operator it is possible to find the unless operator in a systematical way. This can be done using the paper [Aceto *et al.*, 1994] where an algorithm is given to generate a sound and complete axiomatization from a set of operational rules that satisfy a certain SOS format (the so-called GSOS format, see further on).

An interesting point concerning the operational rules of the priority operator and the unless operator is the appearance of negative premises in them. Clearly, such rules do not satisfy the *path* format. Therefore, in this subsection we will make a third journey to the area of general theory on operational semantics. Next, we will generalize the theory that we already treated in subsection 2.2.3. As a running example we take the operational semantics of basic process algebra with priorities. This subsection is based on [Verhoef, 1994a].

In the following definition we generalize the notion of a term deduction system (cf. definition 2.2.18) in the sense that deduction rules may also contain negative premises. [Bloom *et al.*, 1988] formulated the first format with negative premises; it is called the GSOS format. [Groote, 1990b] generalized this substantially and he proposed the so-called *ntyft/ntyxt* format.

**Definition 2.10.4.** A term deduction system is a structure  $(\Sigma, D)$  with  $\Sigma$  a signature and  $D$  a set of deduction rules. The set  $D = D(T_p, T_r)$  is parameterized with two sets, which are called respectively the set of predicate symbols and the set of relation symbols. Let  $s, t$ , and  $u \in O(\Sigma)$ ,  $P \in T_p$ , and  $R \in T_r$ . We call expressions  $Ps$ ,  $\neg Ps$ ,  $tRu$ , and  $t \neg R$  formulas. We call the formulas  $Ps$  and  $tRu$  positive and  $\neg Ps$  and  $t \neg R$  negative. If  $S$  is a set of formulas we write  $PF(S)$  for the subset of positive formulas of  $S$  and  $NF(S)$  for the subset of negative formulas of  $S$ .

A deduction rule  $d \in D$  has the form

$$\frac{H}{C}$$

with  $H$  a set of formulas and  $C$  a positive formula; to save space we will also use the notation  $H/C$ . We call the elements of  $H$  the hypotheses of  $d$  and we call the formula  $C$  the conclusion of  $d$ . If the set of hypotheses of a deduction rule is empty we call such a rule an axiom. We denote an axiom simply by its conclusion provided that no confusion can arise. The notions “substitution”, “var”, and “closed” extend to formulas and deduction rules as expected.

**Example 2.10.5.** A typical example of a term deduction system with negative premises is the operational semantics of  $BPA_{\delta\theta}$ . The term deduction system  $T(BPA_{\delta\theta})$  has as signature that of the equational specification  $BPA_{\delta\theta}$  and its rules are the rules of tables 5, 35, and 36.

Next, we formalize the notion when a formula holds in a term deduction system with negative premises.

**Definition 2.10.6.** Let  $T$  be a term deduction system. Let  $F(T)$  be the set of all closed formulas over  $T$ . We denote the set of all positive formulas over  $T$  by  $PF(T)$  and the negative formulas by  $NF(T)$ . Let  $X \subseteq PF(T)$ . We define when a formula  $\varphi \in F(T)$  holds in  $X$ , notation  $X \vdash \varphi$ :

$$\begin{aligned} X \vdash sRt & \quad \text{if } sRt \in X, \\ X \vdash Ps & \quad \text{if } Ps \in X, \\ X \vdash s \neg R & \quad \text{if } \forall t \in C(\Sigma) : sRt \notin X, \\ X \vdash \neg Ps & \quad \text{if } Ps \notin X. \end{aligned}$$

The purpose of a term deduction system is to define a set of positive formulas that can be deduced using the deduction rules. For instance, if the term deduction system contains only positive formulas then the set of deducible formulas comprises all the formulas that can be proved by a well-founded proof tree. If we allow negative formulas in the premises of a deduction rule it is no longer obvious which set of positive formulas can be deduced using the deduction rules. [Bloom *et al.*, 1988] formulate that

a transition relation must agree with a transition system specification. We will use their notion; it is only adapted in order to incorporate predicates.

**Definition 2.10.7.** Let  $T = (\Sigma, D)$  be a term deduction system and let  $X \subseteq PF(T)$  be a set of positive closed formulas. We say that  $X$  agrees with  $T$  if a formula  $\varphi$  is in  $X$  if and only if there is a deduction rule instantiated with a closed substitution such that the instantiated conclusion equals  $\varphi$  and all the instantiated hypotheses hold in  $X$ . More formally:  $X$  agrees with  $T$  if

$$\varphi \in X \iff \exists H/C \in D, \sigma : V \longrightarrow C(\Sigma) : \sigma(C) = \varphi, \forall h \in H : X \vdash \sigma(h).$$

[Groote, 1990b] showed that if for each rule the conclusions are in some sense more difficult than the premises, there is always a set of formulas that agrees with the given rules. [Verhoef, 1994a] generalized this to the case where predicates come into play. Next, we will formalize this notion that is termed a stratification.

**Definition 2.10.8.** Let  $T = (\Sigma, D)$  be a term deduction system. A mapping  $S : PF(T) \longrightarrow \alpha$  for an ordinal  $\alpha$  is called a stratification for  $T$  if for all deduction rules  $H/C \in D$  and closed substitutions  $\sigma$  the following conditions hold. For all  $h \in PF(H)$  we have  $S(\sigma(h)) \leq S(\sigma(C))$ ; for all  $s \neg R \in NF(H)$  we have for all  $t \in C(\Sigma) : S(\sigma(sRt)) < S(\sigma(C))$ ; for all  $\neg Ps \in NF(H)$  we have  $S(\sigma(Ps)) < S(\sigma(C))$ . We call a term deduction system stratifiable if there exists a stratification for it.

**Remark 2.10.9.** Next, we will give a recipe for finding a stratification. In most cases we can find a stratification (for which the two conditions hold) by measuring the complexity of a positive formula in terms of counting a particular symbol occurring in the conclusion of a rule with negative premises.

**Example 2.10.10.** As an example of the use of the above rule of thumb, we give a stratification for the term deduction system  $T(\text{BPA}_{\delta\theta})$ . The rules containing negative premises have in their conclusion a  $\theta$  or an  $\triangleleft$ . We define a map that counts the number of  $\theta$ 's and the number of  $\triangleleft$ 's as follows: let  $t$  be a closed term with  $n_0$  occurrences of  $\theta$ 's; and  $n_1$  occurrences of  $\triangleleft$ 's then  $S(t \xrightarrow{a} s) = S(t \xrightarrow{a} \sqrt{\phantom{x}}) = n_0 + n_1$ . Now we check the two conditions for the first rule of table 35. Replace each  $x$  and  $x'$  by closed terms  $t$  and  $t'$ . Since the number of  $\theta$ 's plus the number of  $\triangleleft$ 's occurring in  $\theta(t)$  is one greater than the number of  $\theta$ 's plus the number of  $\triangleleft$ 's occurring in  $t$  we are done. The other rules are equally simple.

Next, it is our aim to define a set of positive formulas that agrees with a given term deduction system. Therefore, we will use the following notion. Just think of it as a uniform upper bound to the number of positive premises in a given term deduction system. In general, it is not the least



upper bound.

**Definition 2.10.11.** Let  $V$  be a set. If  $0 \leq |V| < \aleph_0$  we define the degree of  $V$ , denoted by  $d(V)$  to equal  $\omega_0$ . If  $|V| = \aleph_\alpha$  for an ordinal  $\alpha \geq 0$  we define  $d(V) = \omega_{\alpha+1}$ .

Let  $T = (\Sigma, D)$  be a term deduction system. The degree  $d(H/C)$  of a deduction rule  $H/C \in D$  is the degree of its set of positive premises; in a formula:  $d(H/C) = d(PF(H))$ . Let  $\omega_\alpha = \sup\{d(H/C) : H/C \in D\}$ . The degree  $d(T)$  of a term deduction system  $T$  is  $\omega_0$  if  $\alpha = 0$  and  $\omega_{\alpha+1}$  otherwise.

**Example 2.10.12.** It is not hard to see that the degree of our running example is  $\omega_0$ . In fact, we will only treat term deduction systems with degree  $\omega_0$  in this survey. See, for instance, [Klusener, 1993] for rules that contain infinitely many premises.

Next, we will define this set of positive formulas for which it can be shown that it agrees with a given term deduction system. This definition originates from [Groote, 1990b] and is adapted to our situation by [Verhoef, 1994a].

**Definition 2.10.13.** Let  $T = (\Sigma, D)$  be a term deduction system and let  $S : PF(T) \rightarrow \alpha$  be a stratification for an ordinal number  $\alpha$ . We define a set  $T_S \subseteq PF(T)$  as follows:

$$T_S = \bigcup_{i < \alpha} T_i^S, \quad T_i^S = \bigcup_{j < d(T)} T_{i,j}^S.$$

It will be useful to introduce the following notations for certain unions over  $T_i^S$  and  $T_{i,j}^S$ :

$$U_i^S = \bigcup_{i' < i} T_{i'}^S \quad (i \leq \alpha), \quad U_{i,j}^S = \bigcup_{j' < j} T_{i,j'}^S \quad (j \leq d(T)).$$

We drop the sub- and superscripts  $S$  and, for instance, render  $U_i^S$  as  $U_i$  and  $T_S \vdash \varphi$  as  $T \vdash \varphi$ , provided no confusion arises. Now we define for all  $i < \alpha$  and for all  $j < d(T)$  the set  $T_{i,j} = T_{i,j}^S$ :

$$T_{i,j} = \left\{ \varphi \mid S(\varphi) = i, \exists H/C \in D, \sigma : V \rightarrow C(\Sigma) : \sigma(C) = \varphi, \right. \\ \left. \forall h \in PF(H) : U_{i,j} \cup U_i \vdash \sigma(h), \forall h \in NF(H) : U_i \vdash \sigma(h) \right\}.$$

The next theorem is taken from [Verhoef, 1994a] but its proof is essentially the same as a similar theorem of [Groote, 1990b]. It states that for a stratifiable term deduction system the set that we defined above agrees with it. Moreover, this is independent of the choice of the stratification.



**Theorem 2.10.14.** *Let  $T = (\Sigma, D)$  be a term deduction system and let  $S : PF(T) \rightarrow \alpha$  be a stratification for an ordinal number  $\alpha$ . Then  $T_S$  agrees with  $T$ . If  $S'$  is also a stratification for  $T$  then  $T_S = T_{S'}$ .*

**Example 2.10.15.** Since our running example is stratifiable it follows from the above theorem that the term deduction system  $T(\text{BPA}_{\delta\theta})$  determines a transition relation (with predicates) on closed terms.

So, now we only know that when a term deduction system has a stratification there exists some set of positive formulas that agrees with it. Next, we are interested in the conditions under which strong bisimulation equivalence is a congruence relation. Just as in subsection 2.2.3 we define a syntactical restriction on a term deduction system. We will generalize the *path* format to the so-called *panth* format, which stands for “predicates and *ntyft/ntyxt* hybrid format”. The *ntyft/ntyxt* format stems from [Groote, 1990b].

**Definition 2.10.16.** Let  $T = (\Sigma, D)$  be a term deduction system with  $D = D(T_p, T_r)$ . Let in the following  $K, L, M$ , and  $N$  be index sets of arbitrary cardinality, let  $s_k, t_l, u_m, v_n, t \in O(\Sigma)$  for all  $k \in K$ ,  $l \in L$ ,  $m \in M$ , and  $n \in N$ , let  $P_k, P_m, P \in T_p$  be predicate symbols for all  $k \in K$  and  $m \in M$ , and let  $R_l, R_n, R \in T_r$  be relation symbols for all  $l \in L$  and  $n \in N$ .

A deduction rule  $d \in D$  is in *panth* format if it has one of the following four forms:

$$\frac{\{P_k s_k : k \in K\} \cup \{t_l R_l y_l : l \in L\} \cup \{\neg P_m u_m : m \in M\} \cup \{v_n \neg R_n : n \in N\}}{C}$$

- with  $C = f(x_1, \dots, x_n)Rt$ ,  $f \in \Sigma$  an  $n$ -ary function symbol,  $X = \{x_1, \dots, x_n\}$ ,  $Y = \{y_l : l \in L\}$ , and  $X \cup Y \subseteq V$  a set of distinct variables;
- with  $C = xRt$ ,  $X = \{x\}$ ,  $Y = \{y_l : l \in L\}$ , and  $X \cup Y \subseteq V$  a set of distinct variables;
- with  $C = Pf(x_1, \dots, x_n)$ ,  $X = \{x_1, \dots, x_n\}$ ,  $Y = \{y_l : l \in L\}$ , and  $X \cup Y \subseteq V$  a set of distinct variables; or
- with  $C = Px$ ,  $X = \{x\}$ ,  $Y = \{y_l : l \in L\}$ , and  $X \cup Y \subseteq V$  a set of distinct variables.

A term deduction system is in *panth* format if all its rules are.

**Example 2.10.17.** It is not hard to verify that the deduction rules of our running example satisfy the *panth* format.

Next, we define the notion of strong bisimulation for term deduction systems with negative premises. In definition 2.2.27 we gave the positive case. This definition is based on [Park, 1981] and its formulation is taken from [Verhoef, 1994a].

**Definition 2.10.18.** Let  $T = (\Sigma, D)$  be a term deduction system with stratification  $S$  and let  $D = D(T_p, T_r)$ . A binary relation  $B \subseteq C(\Sigma) \times C(\Sigma)$  is called a (strong) bisimulation if for all  $s, t \in C(\Sigma)$  with  $sBt$  the following conditions hold. For all  $R \in T_r$

$$\forall s' \in C(\Sigma) (T_S \vdash sRs' \Rightarrow \exists t' \in C(\Sigma) : T_S \vdash tRt' \wedge s'Bt'),$$

$$\forall t' \in C(\Sigma) (T_S \vdash tRt' \Rightarrow \exists s' \in C(\Sigma) : T_S \vdash sRs' \wedge s'Bt'),$$

and for all  $P \in S_p$

$$T_S \vdash Ps \Leftrightarrow T_S \vdash Pt.$$

The first two conditions are known as the transfer property. Two states  $s$  and  $t \in C(\Sigma)$  are bisimilar if there exists a bisimulation relation containing the pair  $(s, t)$ . If  $s$  and  $t$  are bisimilar we write  $s \sim t$ . Note that bisimilarity is an equivalence relation, called bisimulation equivalence.

At this point we have all the ingredients that we need to formulate the theorem that is interesting for our purpose: the congruence theorem for the *panth* format. It states that in many situations strong bisimulation equivalence is a congruence. The congruence theorem is taken from [Verhoef, 1994a] albeit that there the well-founded subcase is proved. [Fokink, 1994] showed that this condition is not necessary. Thus, we dropped the extra assumption.

**Theorem 2.10.19.** *Let  $T = (\Sigma, D)$  be a stratifiable term deduction system in panth format. Then strong bisimulation equivalence is a congruence for all function symbols.*

**Example 2.10.20.** Since the deduction rules of our running example are in *panth* format and since the term deduction system has a stratification, we find with the congruence theorem that strong bisimulation equivalence is a congruence.

According to the above example we find that the quotient of the closed  $\text{BPA}_{\delta\theta}$  terms modulo bisimulation equivalence is well-defined; this means that the operators of  $\text{BPA}_{\delta\theta}$  can be defined on this quotient. By a straightforward proof we can show that it is a model of  $\text{BPA}_{\delta\theta}$ .

We postpone the proof of the completeness of  $\text{BPA}_{\delta\theta}$  until we have shown that it is a conservative extension of  $\text{BPA}_\delta$ .

### 2.10.2 Conservativity

In this subsection we take care of the conservativity of  $\text{BPA}_{\delta\theta}$  over BPA. We are used to proving this via the conservativity theorem for the *path* format but since the operational rules of  $\text{BPA}_{\delta\theta}$  do not fit this format, we cannot simply apply this theorem. Just as with the conservativity of  $\text{BPA}_{\delta\theta}$  (see subsection 2.10.1) over BPA we will generalize below the theory that we already treated on conservativity—yet another trip into the general theory

on operational semantics. This time we will mainly extend the theory of section 2.4.1 so that we can also deal with negative premises. This subsection is based on [Verhoef, 1994b].

Since we treated some theory on negative premises and some theory on conservative extensions, their combination will be not too much work. We have to update the notions of pure, well-founded, and operationally conservative extension. Then only the operationally conservative extension theorem for the *path* format needs a little modification.

Below, we give the update of the notion pure. It was defined in the positive case in definition 2.2.30.

**Definition 2.10.21.** A deduction rule containing negative premises is pure if this rule is already pure when the negative premises are discarded. A term deduction system with negative premises is pure if all its deduction rules are pure.

**Example 2.10.22.** It is not hard to see that the term deduction system  $T(\text{BPA}_{\delta\theta})$  is pure.

Now, we update the definition of well-founded. This notion is defined in definition 2.4.13 for the positive case. The update is in the same vein as the one for the purity.

**Definition 2.10.23.** Let  $T = (\Sigma, D)$  be a term deduction system and let  $F$  be a set of formulas. The variable dependency graph of  $F$  is a directed graph with variables occurring in  $F$  as its nodes. The edge  $x \rightarrow y$  is an edge of the variable dependency graph if and only if there is a positive relation  $tRs \in F$  with  $x \in \text{var}(t)$  and  $y \in \text{var}(s)$ .

The set  $F$  is called well-founded if any backward chain of edges in its variable dependency graph is finite. A deduction rule is called well-founded if its set of hypotheses is so. A term deduction system is called well-founded if all its deduction rules are well-founded.

**Example 2.10.24.** It is not hard to see that the term deduction system  $T(\text{BPA}_{\delta\theta})$  is well-founded.

Next, we update the notion of an operationally conservative extension. Also this definition does not look very different from its positive counterpart. Note that in the positive case proofs are well-founded trees, whereas in the negative case we use the notion of agreeing with. More information on this can be found in subsection 2.10.1. We also refer to this subsection for the definition of stratifiability.

**Definition 2.10.25.** Let  $T^i = (\Sigma_i, D_i)$  be term deduction systems with  $T = (\Sigma, D) := T^0 \oplus T^1$  defined. Let  $D = D(T_p, T_r)$ . The term deduction system  $T$  is called an operationally conservative extension of  $T^0$  if it is stratifiable and for all  $s, u \in C(\Sigma_0)$ , for all relation symbols  $R \in T_r$  and predicate symbols  $P \in T_p$ , and for all  $t \in C(\Sigma)$  we have

$$T_S \vdash sRt \iff T_{S^0}^0 \vdash sRt$$

and

$$T_S \vdash Pu \iff T_{S^0}^0 \vdash Pu,$$

where  $S$  is a stratification for  $T$  and  $S^0$  is a stratification for  $T^0$  (take for instance  $S^0$  to be the restriction of  $S$  to positive formulas of  $T^0$ ).

Now we have all the updates of the definitions that we need in order to state the operationally conservative extension theorem for the *panth* format. The following theorem is taken from [Verhoef, 1994b].

**Theorem 2.10.26.** *Let  $T^0 = (\Sigma_0, D_0)$  be a pure well-founded term deduction system in panth format. Let  $T^1 = (\Sigma_1, D_1)$  be a term deduction system in panth format. If there is a conclusion  $sRt$  or  $Ps$  of a rule  $d_1 \in D_1$  with  $s = x$  or  $s = f(x_1, \dots, x_n)$  for an  $f \in \Sigma_0$ , we additionally require that  $d_1$  is pure, well-founded,  $t \in O(\Sigma_0)$  for premises  $tRy$  of  $d_1$ , and that there is a positive premise containing only  $\Sigma_0$  terms and a new relation or predicate symbol. Now if  $T = T^0 \oplus T^1$  is defined and stratifiable then  $T$  is an operationally conservative extension of  $T_0$ .*

**Example 2.10.27.** In subsection 2.10.1 we already showed that the term deduction system that belongs to  $\text{BPA}_{\delta\theta}$  is stratifiable. It is easy to verify the other conditions of the above theorem so we may conclude that  $\text{BPA}_{\delta\theta}$  is an operationally conservative extension of BPA.

In the above example we have shown the operational conservativity of  $\text{BPA}_{\delta\theta}$  over BPA. We are in fact interested in the equational conservativity. The other theorems, in particular the equationally conservative extension theorem, that we treated in subsection 2.4.1, do not need any updates, since in those theorems we only refer to term deduction systems and we do not specify which ones. [Verhoef, 1994b] showed that these theorems hold for term deduction systems with negative premises.

So, we can formulate and prove the following theorem.

**Theorem 2.10.28.** *The equational specification  $\text{BPA}_{\delta\theta}$  is an equationally conservative extension of BPA.*

**Proof.** Straightforward: check the conditions of theorem 2.4.24 and use example 2.10.27. ■

Now that we have the conservativity result, the completeness of  $\text{BPA}_{\delta\theta}$  follows more or less from the completeness of  $\text{BPA}_\delta$ . We will see this in the following theorem.

**Theorem 2.10.29.** *The equational specification  $\text{BPA}_{\delta\theta}$  is a complete axiomatization of the set of closed  $\text{BPA}_{\delta\theta}$  terms modulo bisimulation equivalence.*



**Proof.** Easy: use theorem 2.4.26. Note that the priority operator can be eliminated; see theorem 2.10.3. ■

### 2.10.3 Extensions of $\text{BPA}_{\delta\theta}$

In this subsection we discuss extensions of  $\text{BPA}_{\delta\theta}$  with the notions of recursion, projections, renaming, and/or the encapsulation operator, and the state operator. In fact, all extensions but the one with recursion can be obtained just as for the BPA or  $\text{BPA}_{\delta}$  case.

*Recursion* The problem with the extension of  $\text{BPA}_{\delta\theta}$  with recursion is purely technical. Since there are negative premises in the operational characterization of the priority and unless operators, we introduced the notion of a stratification to ensure that the semantical rules indeed define a transition relation. We recall that in example 2.10.10 we give a stratification for the operational semantics of  $\text{BPA}_{\delta\theta}$ . The map defined there counts the total of occurrences of  $\theta$  and  $\triangleleft$ . This approach no longer works in the presence of the operational rules for recursion that we presented in table 6. We illustrate this with a simple example. Suppose that we have the following recursive specification:

$$E = \{X = a \cdot X + \theta(a)\}.$$

In this case, the operational rule takes the form

$$\frac{a \cdot \langle X|E \rangle + \theta(a) \xrightarrow{a} \langle X|E \rangle}{\langle X|E \rangle \xrightarrow{a} \langle X|E \rangle}.$$

So with the above stratification we have that the stratification of the premise is *not* less than or equal to the stratification of the conclusion. To solve this problem we use infinite ordinals. We adapt the stratification as follows:

$$S(t \xrightarrow{a} t') = \omega \cdot n + m,$$

where  $n$  is the number of *unguarded* occurrences of  $\langle$  and  $m$  is the total occurrences of  $\theta$  and occurrences of  $\triangleleft$  (so the  $m$  part is the original stratification). With the modified stratification, the problem is solved. We leave it as an exercise to the reader to check the details.

*Projection* The extension of  $\text{BPA}_{\delta\theta}$  with projection is obtained in the same way as this extension for BPA; see subsection 2.4.

*Renaming and encapsulation* It is straightforward to extend the equational specification  $\text{BPA}_{\delta\theta}$  with renaming operators or the encapsulation operator; cf. subsection 2.7.3.

*State operator* The extension of the theory  $\text{BPA}_{\delta\theta}$  with either the simple or extended state operator is obtained in the same way as for the theory BPA; see subsections 2.8 and 2.9.



*Inconsistent combinations* Remarkably, if we combine recursion with  $\text{BPA}_{\delta\theta}$  plus renaming operators we will find an inconsistency. [Groote, 1990b] gives the following example. Take a renaming function  $f$  such that

- $f(b) = a$ ,
- $f(a) = c$ ,
- $f(d) = d$  for all  $d \in A \setminus \{a, b\}$ .

Consider the recursive equation

$$X = \theta(\rho_f(X) + b).$$

Now it can be shown that

$$X \xrightarrow{b} \checkmark \iff X \not\xrightarrow{b} \checkmark$$

if we take  $a > b$  as the partial ordering on the atomic actions.

Observe also that the combination of recursion with  $\text{BPA}_{\delta\theta}$  plus state operators is inconsistent since state operators are a generalization of renamings.

## 2.11 Basic process algebra with iteration

In this subsection we extend basic process algebra with an iterative construct. This construct is, in fact, Kleene's star operator, a binary infix operator denoted  $*$ . We will call this operator *Kleene's binary star operator*, since there are two versions of Kleene's star operator: one unary and one binary. The binary construct originates from [Kleene, 1956] and its more commonly known unary version is due to [Copi *et al.*, 1958]. This subsection is based on the papers [Bergstra *et al.*, 1994b] and [Fokkink and Zantema, 1994].

We want to note that using iteration we can also define infinite processes. We already discussed recursion, the standard way to define infinite processes, in subsection 2.3. The advantage of the approach that we explain in this subsection is that there is no need for proof rules like the recursive definition principle or the recursive specification principle, to guarantee that a recursive specification has a possibly unique solution. In this setting, the recursive construct is just some binary operator that we may add to a process language.

*The theory* The equational specification  $\text{BPA}^*$  consists of the signature of BPA and a binary infix operator  $*$ , called Kleene's binary star operator. Its equations are the ones of BPA plus the axioms in table 37.

We will comment on these axioms. The first one BKS1 is the defining equation for the star operator that [Kleene, 1956] gives in the context of finite automata. Only the notation is adapted to the present situation.

**Table 37.** The axioms defining Kleene's binary star operator.

$x(x^*y) + y = x^*y$	BKS1
$x^*(yz) = (x^*y)z$	BKS2
$x^*\left(y((x+y)^*z) + z\right) = (x+y)^*z$	BKS3

The second equation originates from [Bergstra *et al.*, 1994b]; it is a simple equation needed for the completeness. The third axiom, BKS3, is more sophisticated; it stems from [Troeger, 1993]. Troeger used this equation for a slightly different process specification formalism.

Next, we will show some properties that can be derived from the equational specification  $\text{BPA}^*$ . For instance, if we apply Kleene's axiom to the first term in the display below we find a term to which we can apply Troeger's axiom with  $x+y$  substituted for  $y$ . Thus, this yields the following identity:

$$\begin{aligned} x^*((x+y)^*z) &= x^*((x+y)((x+y)^*z) + z) \\ &= (x+y)^*z. \end{aligned}$$

The next identity expresses that applying the star operator in a nested way for the same process reduces to applying it once. First, we apply Kleene's axiom, then we use the idempotence of the alternative composition, then we use Troeger's identity, and then one application of idempotence finishes the calculation. We display this below.

$$\begin{aligned} x^*(x^*y) &= x^*(x(x^*y) + y) \\ &= x^*(x((x+x)^*y) + y) \\ &= (x+x)^*y \\ &= x^*y. \end{aligned}$$

*Semantics* We give the semantics of the equational specification  $\text{BPA}^*$  by means of a term deduction system  $T(\text{BPA}^*)$ . Its signature is the signature of  $\text{BPA}^*$ . Its deduction rules are the rules for  $\text{BPA}$  that we met many times before (see table 5) plus the rules that characterize Kleene's binary star operator. We list them in table 38.

**Theorem 2.11.1.** *The set of closed  $\text{BPA}^*$  terms modulo strong bisimulation equivalence is a model of  $\text{BPA}^*$ .*

**Proof.** Since bisimulation equivalence is a congruence, we only need to check the soundness of the axioms of  $\text{BPA}^*$ . The first five axioms are already

treated in the soundness theorem for BPA (see 2.2.33). So it suffices to prove the soundness of the three remaining equations. The case BKS1 is proved analogously to the case A1: take as relation the pair  $(x(x^*y) + y, x^*y)$  and the diagonal. Now it is not hard to show that this is a bisimulation relation. For the equation BKS2 we have the following relation: relate all terms of the form  $x^*(yz)$  with  $(x^*y)z$ ; relate each term of the form  $x' \cdot (x^*(y \cdot z))$  with  $(x' \cdot (x^*y)) \cdot z$ ; and relate each term with itself. We leave it to the reader to verify that this relation is a bisimulation relation. The verification of the soundness of Troeger's axiom is obtained analogously to the verification of equation BKS2. ■

It is easy to see that  $\text{BPA}^*$  is a conservative extension of BPA; see subsection 2.4.1. However, we cannot eliminate Kleene's binary star operator. See subsection 2.14 where we discuss expressivity results. This can be easily seen as follows. Call a term deduction system  $T$  operationally terminating if there are no infinite reductions

$$T \vdash s_0 R_0 s_1, T \vdash s_1 R_1 s_2, \dots$$

possible. It is easy to see by inspection of the operational rules for BPA that its term deduction system is operationally terminating (cf. lemma 2.2.34 where a "weight" function is defined). It is also easy to see that the term deduction system belonging to  $\text{BPA}^*$  is not operationally terminating. We have, for instance, the infinite reduction

$$a^*b \xrightarrow{a} a^*b \xrightarrow{a} a^*b \xrightarrow{a} \dots$$

Now suppose that Kleene's binary star operator can be eliminated in favour of the operators of BPA. Then  $a^*b$  must be bisimilar to a BPA term, say  $t$ . Because of the bisimilarity with  $a^*b$  we must have that  $t$  can mimic the above steps that  $a^*b$  is able to perform. So  $t$  must have an infinite reduction. This contradicts the fact that the semantics of BPA is operationally terminating.

As a corollary, we cannot use the completeness theorem 2.4.26 to prove the completeness of  $\text{BPA}^*$ . The proof that our axiomatization is nevertheless complete is due to [Fokkink and Zantema, 1994] and is beyond the

**Table 38.** Operational rules for Kleene's binary star operator.

$\frac{x \xrightarrow{a} x'}{x^*y \xrightarrow{a} x' \cdot (x^*y)}$	$\frac{x \xrightarrow{a} \surd}{x^*y \xrightarrow{a} x^*y}$
$\frac{y \xrightarrow{a} y'}{x^*y \xrightarrow{a} y'}$	$\frac{y \xrightarrow{a} \surd}{x^*y \xrightarrow{a} \surd}$

**Table 39.** The axioms defining the discrete time unit delay.

$\sigma_d(x) + \sigma_d(y) = \sigma_d(x + y)$	DT1
$\sigma_d(x) \cdot y = \sigma_d(x \cdot y)$	DT2

scope of this chapter. The reason for this is that the proof makes use of a sophisticated term rewriting analysis. Below, we will list their main result.

**Theorem 2.11.2.** *The equational specification  $\text{BPA}^*$  is a complete axiomatization with respect to strong bisimulation equivalence.*

**Proof.** See the paper [Fokkink and Zantema, 1994]. ■

*Extensions of  $\text{BPA}^*$*  The extension of  $\text{BPA}^*$  with deadlock ( $\text{BPA}_\delta^*$ ) is as usual. This system is obtained by taking the syntax of  $\text{BPA}_\delta$  plus Kleene's binary operator  $*$ . The axioms of  $\text{BPA}_\delta^*$  are the ones of  $\text{BPA}^*$  plus those for deadlock.

Since  $\text{BPA}_\delta^*$  is more expressive (see subsection 2.14) than  $\text{BPA}^*$  we cannot use the usual machinery to prove basic properties such as completeness. There is no completeness result for the  $\text{BPA}_\delta^*$  system so we will not discuss the extensions of  $\text{BPA}^*$  (or  $\text{BPA}_\delta^*$ ) with the notions that we usually extend our systems with. Moreover, at the time of writing this survey the only studied extensions of  $\text{BPA}^*$  are those with abstraction, fairness principles, deadlock, and parallel constructs. We will discuss some of these extensions after we have introduced such parallel constructs.

## 2.12 Basic process algebra with discrete relative time

Now, we treat an extension of BPA with a form of discrete relative time; we abbreviate this as  $\text{BPA}_{\text{dt}}$ . We speak of discrete time since the system works with so-called time slices. It is called relative since the system refers to the current time slice, the next time slice, and so on.  $\text{BPA}_{\text{dt}}$  stems from [Baeten and Bergstra, 1992a]. For other approaches to discrete time process algebra we refer to [Moller and Tofts, 1990] and [Nicollin and Sifakis, 1994].

*Theory* The equational specification  $\text{BPA}_{\text{dt}}$  has as its signature the one of BPA and a unary function called discrete time unit delay, which is denoted  $\sigma_d$ . The  $\sigma$  is some fixed symbol, which is a measure for the delay. The axioms of  $\text{BPA}_{\text{dt}}$  are the ones of BPA that we listed in table 1 plus the equations defining the discrete time unit delay; see table 39.

We denote the atomic action  $a$  in the current time slice by  $\underline{a}$ . We distinguish  $\underline{a}$  from  $a$  because also other embeddings of BPA into  $\text{BPA}_{\text{dt}}$  are possible, where  $a$  is interpreted as  $a$  occurs at some time, that is, we have  $a = \underline{a} + \sigma_d(a)$ . The intended interpretation of the unary operator  $\sigma_d(x)$  is that it pushes a process  $x$  to the next time slice. The length of a time

slice is measured with the positive real  $\sigma$ . This is operationally expressed by the rule  $\sigma_d(x) \xrightarrow{\sigma} x$ , where  $\xrightarrow{\sigma}$  is a special relation that describes the pushing behaviour. Note that the label  $\sigma$  is *not* part of the signature of  $\text{BPA}_{dt}$ .

Axiom DT1 is called the “time factorizing axiom”. It expresses that the passage of time by itself cannot determine a choice. We note that the form of choice here is called “strong choice” (the other two approaches mentioned above have weak choice), so in  $\underline{a} + \sigma_d(\underline{b})$  both  $a$  in the current time slice and  $b$  in the next time slice are possible. We do have in the closed term above that by moving to the next time slice, we disable  $\underline{a}$ .

Next, we will show that the term rewriting system associated to  $\text{BPA}_{dt}$  is terminating. Although this result has importance of its own, we cannot use it to prove an elimination result. For the discrete time unit delay cannot be eliminated.

**Theorem 2.12.1.** *The term rewriting system that is associated to  $\text{BPA}_{dt}$  is strongly normalizing. This system consists of the rules in tables 2 and 40.*

**Proof.** We use the method of the recursive path ordering that we treated in subsection 2.2.2. Take as precedence for the operations in the signature  $\cdot > + > \sigma_d$  and give the sequential composition the lexicographical status for the first argument. As an example, we treat RDT2.

$$\begin{aligned}
 \sigma_d(x) \cdot y &>_{lpo} \sigma_d(x) \cdot^* y \\
 &>_{lpo} \sigma_d(\sigma_d(x) \cdot^* y) \\
 &>_{lpo} \sigma_d(\sigma_d^*(x) \cdot y) \\
 &>_{lpo} \sigma_d(x \cdot y).
 \end{aligned}$$

The other rule is dealt with just as simply. ■

Now that we know that the term rewriting system associated to  $\text{BPA}_{dt}$  is terminating, we discuss what form the normal forms can take. Following [Baeten and Bergstra, 1992a], we define these normal forms, called basic terms.

**Definition 2.12.2.** In order to define inductively the set of basic terms, we need the auxiliary notion of an  $A$ -basic term: a  $\text{BPA}_{dt}$  term with no leading  $\sigma_d$ . We define both notions simultaneously.

- every  $A$ -basic term is a basic term;

**Table 40.** The rewrite rules for the discrete time unit delay.

$\sigma_d(x) + \sigma_d(y) \rightarrow \sigma_d(x + y)$	RDT1
$\sigma_d(x) \cdot y \rightarrow \sigma_d(x \cdot y)$	RDT2



- for each  $a \in A$ ,  $\underline{a}$  is an  $A$ -basic term;
- if  $a \in A$  and  $t$  is a basic term, then  $\underline{a} \cdot t$  is an  $A$ -basic term;
- if  $t$  and  $s$  are  $A$ -basic terms, then  $t + s$  is an  $A$ -basic term;
- if  $t$  is a basic term, then  $\sigma_d(t)$  is a basic term;
- if  $t$  is an  $A$ -basic term and  $s$  is a basic term, then  $t + \sigma_d(s)$  is a basic term.

Next, we formulate some facts from [Baeten and Bergstra, 1992a]. They can be easily proved.

**Theorem 2.12.3.**

- Let  $t$  be a closed  $\text{BPA}_{\text{dt}}$  term. Then there exists a basic term  $s$  such that  $\text{BPA}_{\text{dt}} \vdash t = s$ .
- An  $A$ -basic term takes the form:  $\sum_{i < n} \underline{a}_i \cdot t_i + \sum_{j < m} \underline{b}_j$  with  $n+m > 0$ ,  $a_i, b_j \in A$ , and  $t_i$  basic.
- A basic term is either an  $A$ -basic term or of the form  $t + \sigma_d(s)$  with  $t$  and  $s$   $A$ -basic terms.

*Semantics* Next, we formally define the semantics by way of a term deduction system for  $\text{BPA}_{\text{dt}}$ . The signature of this system consists of the one for the equational specification  $\text{BPA}_{\text{dt}}$ . The deduction rules are those for  $+$  and  $\cdot$  of  $\text{BPA}$  in table 1, and the rules for constants and the discrete time unit delay in table 41.

Note the appearance of negative premises in the operational rules. By means of the theory that we discussed in subsection 2.10.1, we can find that this system indeed defines a set of positive formulas. We recall that we, therefore, have to find a stratification. Let  $n$  be the number of  $+$  signs that occurs in a closed  $\text{BPA}_{\text{dt}}$  term  $t$ . Then we define a stratification  $S$  by assigning to  $t \xrightarrow{\sigma} s$  and to  $t \xrightarrow{\sigma} \surd$  the number  $n$ . For the other formulas  $\varphi$  we simply define  $S(\varphi) = 0$ . It is not hard to see that this function is a stratification. So the term deduction system is well-defined.

It is easy to see that the deduction rules are in *panth* format (see subsection 2.10.1), so we find with theorem 2.10.19 that strong bisimulation equivalence is a congruence.

**Theorem 2.12.4.** *The equational specification  $\text{BPA}_{\text{dt}}$  is a sound axiomatization.*

**Table 41.** The operational semantics for the discrete time unit delay.

$\underline{a} \xrightarrow{a} \surd$	$\sigma_d(x) \xrightarrow{\sigma} x$	$\frac{x \xrightarrow{\sigma} x'}{x \cdot y \xrightarrow{\sigma} x' \cdot y}$
$\frac{x \xrightarrow{\sigma} x', y \xrightarrow{\sigma} y'}{x + y \xrightarrow{\sigma} x' + y'}$	$\frac{x \xrightarrow{\sigma} x', y \not\xrightarrow{\sigma}}{x + y \xrightarrow{\sigma} x'}$	$\frac{x \not\xrightarrow{\sigma}, y \xrightarrow{\sigma} y'}{x + y \xrightarrow{\sigma} y'}$

tization of the set of closed  $\text{BPA}_{\text{dt}}$  terms modulo strong bisimulation equivalence.

**Proof.** We can give the soundness proof along the usual lines: the soundness of the BPA axioms is already done and the soundness of the equations DT1–2 can be obtained just as the case A1. ■

**Theorem 2.12.5.** *The equational specification  $\text{BPA}_{\text{dt}}$  is a complete axiomatization of the set of closed  $\text{BPA}_{\text{dt}}$  terms modulo strong bisimulation equivalence.*

**Proof.** Usually, we prove the completeness with theorem 2.4.26. However, in this case we cannot apply our routine approach. This is due to the fact that the discrete time unit delay cannot be eliminated; for instance, the term  $\sigma_a(\underline{a})$  cannot be reduced any further. So we cannot apply our completeness theorem 2.4.26, since there we assume that extra operators can be eliminated.

To prove the completeness we follow [Baeten and Bergstra, 1992a]. Their idea is to define a bijective mapping between  $\text{BPA}_{\text{dt}}$  terms and BPA terms; the completeness of  $\text{BPA}_{\text{dt}}$  now follows from the completeness of BPA. Next, we will work out their idea.

The equational specification  $\text{BPA}_{\text{dt}}$  is parameterized with a set of atomic actions  $A$ ; we write  $\text{BPA}_{\text{dt}}(A)$ . Similarly, the theory BPA is parameterized in this way. Since there is mostly no confusion with which set our systems are equipped, we omit them often—but not in this case, since we parameterize BPA with  $A_\sigma = A \cup \{\sigma\}$ , where  $\sigma \notin A$  is an atomic action (with a suggestive name). So let  $\varphi$  from  $\text{BPA}_{\text{dt}}(A)$  to  $\text{BPA}(A_\sigma)$  be inductively defined as follows:

- $\varphi(\underline{a}) = a$
- $\varphi(x + y) = \varphi(x) + \varphi(y)$
- $\varphi(x \cdot y) = \varphi(x) \cdot \varphi(y)$
- $\varphi(\sigma_a(x)) = \sigma \cdot \varphi(x)$ .

Now, suppose that we have two bisimilar  $\text{BPA}_{\text{dt}}$  terms  $s$  and  $t$ . With the aid of theorem 2.12.3 we may assume that  $s$  and  $t$  are basic terms. So we find that  $\varphi(s)$  and  $\varphi(t)$  are also bisimilar. With the completeness theorem 2.2.35 for BPA we find that  $\text{BPA}(A_\sigma) \vdash \varphi(s) = \varphi(t)$ . Using the inverse mapping of  $\varphi$ , we can mimic each step of this proof by a step in  $\text{BPA}_{\text{dt}}$ . So we find that  $\text{BPA}_{\text{dt}} \vdash s = t$ . ■

Next, we formulate a conservativity result for  $\text{BPA}_{\text{dt}}$ .

**Theorem 2.12.6.** *The equational specification  $\text{BPA}_{\text{dt}}$  is a conservative extension of BPA (using  $\underline{a}$  instead of  $a$ ).*

**Proof.** This can be easily shown using the theory that we discussed in subsection 2.10.2.

We note that this result is due to [Verhoef, 1994b]. ■

### 2.12.1 Extensions of $\text{BPA}_{\text{dt}}$

We will only discuss the extension of  $\text{BPA}_{\text{dt}}$  with deadlock and recursion. We will not treat the other extensions that we usually have. The reason for this is that at the time of writing this survey these have not been formulated.

First, we will discuss how to extend  $\text{BPA}_{\text{dt}}$  with deadlock and then we discuss the extension with recursion.

*Deadlock* We can extend  $\text{BPA}_{\text{dt}}$  with deadlock in the usual way. We abbreviate this equational specification as  $\text{BPA}_{\delta\text{dt}}$ . The axioms for  $\underline{\delta}$  are the usual ones for deadlock; see table 10. The termination proof is a combination of these proofs for  $\text{BPA}_{\text{dt}}$  and  $\text{BPA}_{\delta}$ . The notion of a basic term needs a little modification:  $\underline{\delta}$  is an  $A$ -basic term. The operational semantics is the same as the one for  $\text{BPA}_{\text{dt}}$ . The soundness and completeness are proved along the same lines as the case  $\text{BPA}_{\text{dt}}$ . The conservativity of  $\text{BPA}_{\delta\text{dt}}$  over  $\text{BPA}_{(\delta)}$  is obtained as usual.

*Recursion* The extension of  $\text{BPA}_{\text{dt}}$  with recursion has the same technical problem as the extension of  $\text{BPA}_{\delta\theta}$  with recursion. We recall that the problem is that we need to define a new stratification on the operational rules of  $\text{BPA}_{\text{dt}}$  with recursion in order to guarantee that the transition relation is well-defined. For a solution we refer to subsection 2.10.3 where extensions of  $\text{BPA}_{\delta\theta}$  are discussed.

## 2.13 Basic process algebra with other features

When we want to describe parallel or distributed systems, the most important extensions are the ones with some form of parallel composition. We devote section 3 to such extensions. Below, we list a number of other extensions that we will not cover in this survey. We remark that this list is incomplete and in random order.

*Abstraction* In this survey we only treat concrete process algebra, hence any extension of the systems that we discuss with some notion of abstraction will not be covered by this survey. For more information on process algebras that incorporate abstraction we mention [Bergstra and Klop, 1985] that treats an extension of BPA with abstraction. Other systems that feature abstraction are CCS [Milner, 1980; Milner, 1989], Hennessy's system [Hennessy, 1988], and CSP [Hoare, 1985]. We note that the latter two systems are not extensions of BPA but treat basic notions in a different way. But also CCS is not an extension of BPA because there is no sequential composition in CCS.

There are many other process algebras (with abstraction) such as CIR-CAL [Milne, 1983], MEIJE [Austry and Boudol, 1984], SCCS [Milner, 1983], and the  $\pi$ -calculus [Milner *et al.*, 1992] to mention some.

*Backtracking* A well-known notion in logic programming is backtracking. [Bergstra *et al.*, 1994c] extended process algebra with this notion. They discuss an algebraic description of backtracking by means of a binary operator. For more details on this extension we refer to [Bergstra *et al.*, 1994c].

*Combinatory logic* In [Bergstra *et al.*, 1994a], process algebra is extended with combinatory logic. An interesting point of this combination is the possibility to verify the well-known alternating bit protocol without any conditional axiom, that is, the verification is purely equational. For more information on this combination and the equational verification we refer to [Bergstra *et al.*, 1994a].

*Real-time* In recent years, much effort has been spent on the extension of several process algebras with a notion of time. We discuss in this survey just one such extension: process algebra with relative discrete time. However, there are many more (concrete) extensions present in the literature. We mention the distinction between relative and absolute time, and the choice of the time domain: discrete or dense. We refer to [Klusener, 1993] for more information on real time process algebra in many and diverse forms.

*Real-space* In [Baeten and Bergstra, 1993] a form of real time process algebra is extended with real space. The paper surveys material from former reports on this topic. We refer the interested reader to [Baeten and Bergstra, 1993] for more information.

*Nesting* In this survey, we discuss the extension of process algebra with iteration, or Kleene's binary star. An extension that we do not discuss is one with an operator called the nesting operator. Like Kleene's binary star operator, the nesting operator also is a recursive operator (though it defines irregular recursion). We refer to [Bergstra *et al.*, 1994b] for more information on this topic.

*Signals* In [Baeten and Bergstra, 1992b] process algebra is extended with stable signals. These are attributes of states of a process. They introduce a signal insertion and a signal termination operator to be able to describe signals with a certain duration. A typical example that can be described with this theory is a traffic light system. For more information on the extension with signals we refer to [Baeten and Bergstra, 1992b].

*Conditionals* An extension with conditionals or guards can be found in the just mentioned paper [Baeten and Bergstra, 1992b]. They introduce an if-then-else operator in the notation of [Hoare *et al.*, 1987]. [Baeten and Bergstra, 1992b] also introduce a variant of this conditional operator, called the guarded command that originates from [Baeten *et al.*, 1991]. [Groote and Ponse, 1994] developed a substantial amount of theory for a similar conditional construct called a guard.

For more information on the extension of BPA with conditional constructs we refer to the above papers.



*Invariants and assertions* Often, it is useful to have a connection between algebraic expressions and expressions in a logical language. Logical formulas can be used to express invariants (see [Bezem and Groote, 1994]) or as assertions (see [Ponse, 1991]).

*Probabilities* Often, systems exhibit behaviour that is probabilistic or statistical in nature. For example, one may observe that a faulty communication link drops a message 2% of the time. Algebraic formulations of probabilistic behaviour can be found in [Baeten *et al.*, 1992], [Giacalone *et al.*, 1990], [Larsen and Skou, 1992], and [Tofts, 1990], to mention some.

## 2.14 Decidability and expressiveness results in BPA

In this subsection we briefly mention decidability and expressiveness issues for the family of process algebras that we have introduced thus far.

### 2.14.1 Decidability

In our case, the decidability problems concern the question whether or not two finitely specified processes in, for instance  $\text{BPAREC}$ , are bisimilar; see [Baeten *et al.*, 1993], [Caucal, 1990], and [Christensen *et al.*, 1992]. Informally, we refer to this as the question whether or not  $\text{BPAREC}$  is decidable. It turns out that  $\text{BPAREC}$  is decidable for all guarded processes; see [Christensen *et al.*, 1992]. For almost all extensions of  $\text{BPAREC}$  the decidability problem is open. Only for some extensions of  $\text{BPAREC}$  with the state operator we have some information at the time of writing this survey. We refer the interested reader to [Baeten and Bergstra, 1991] and [Blanco, 1995] for more details on the systems  $\lambda(\text{BPA}_{\delta\text{rec}})$  and  $\text{BPA}_{\delta\lambda\text{rec}}$  and their decidability problems.

The following theorem is taken from [Christensen *et al.*, 1992]. The proof of this theorem is beyond the scope of this survey.

**Theorem 2.14.1.** *Bisimulation equivalence is decidable for all  $\text{BPAREC}$  processes that can be specified with a finite guarded recursive specification.*

### 2.14.2 Expressiveness

For the family of systems that we introduced it is natural to address the question of expressivity. The result that is known states that  $\text{BPAREC}$  can express non-regular processes. So, we first need to know what exactly are regular processes. This well-known definition is formulated below and is taken from [Baeten and Weijland, 1990]. Roughly, a process is regular if it has a finite graph.

First, we define when a process is regular in some model.

**Definition 2.14.2.** Let  $x$  be a process in some model  $M$  of  $\text{BPAREC}$ . Define the relations  $\cdot \xrightarrow{a} \cdot$  on this model as follows:

- $x \xrightarrow{a} y \iff M \models x = x + ay,$



$$\bullet \ x \xrightarrow{a} \surd \iff M \models x = x + a.$$

A process  $y$  is called a subprocess of  $x$  if  $y$  is reachable from  $x$ ; reachability means that there is a path of the following form that begins in  $x$  and ends in  $y$ :

$$x \xrightarrow{a_1} x_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} y.$$

See also the definition of reachability in a term deduction system 2.2.22.

We say that  $x$  is a regular process (for the model  $M$ ) if  $x$  has only finitely many subprocesses.

Next, we define when a guarded recursive specification is linear. It will turn out that a regular process can always be specified by a finite linear specification.

**Definition 2.14.3.** Let  $E$  be a recursive specification with variables from the set  $V$ . The specification  $E$  is called linear if every recursion equation in  $E$  is of the form:

$$X = \sum_{i < n} a_i \cdot X_i + \sum_{j < m} b_j,$$

for certain atomic actions  $a_i$  and  $b_j$  and variables  $X, X_i \in V$  ( $n + m > 0$  and  $n, m \in \mathbb{N}$ ).

We call a recursion equation linear if it takes the above form. Note that every linear specification is guarded.

**Lemma 2.14.4.** *Let  $M$  be a model of BPArec. A process  $x$  is regular for  $M$  if and only if there exists a finite linear specification with  $x$  as solution.*

**Proof.** Sketch. We can turn each model into a graph model with definition 2.14.2. Now given a regular process  $x$ , we turn it into a finite graph. This graph determines a finite linear specification of which  $x$  is a solution.

Vice versa, let  $E$  be a finite linear specification. We can easily associate a finite graph to  $E$ , which in turn represents a regular process. (For instance, in the next example we turn a recursive specification into a graph using the above method.) ■

Next, we show that there is a non-regular process that is finitely expressible in the theory BPArec, namely a counter.

**Example 2.14.5.** Consider the following guarded recursive specification. We call the process  $C$  a counter.

$$\begin{aligned} C &= T \cdot C \\ T &= plus \cdot T' \\ T' &= minus + T \cdot T'. \end{aligned}$$

We give the deduction graph (see definition 2.2.23) of  $C$  in figure 6. Note that we use  $+$  for *plus* and  $-$  for *minus*.



Fig. 6. The deduction graph of the counter  $C$ .

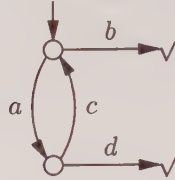


Fig. 7. The deduction graph of a regular process.

It is well known that the counter is a non-regular process. It has infinitely many distinct states, since for each  $n$  there is a state where  $n$  consecutive *minus* steps can be executed but not  $n + 1$ .

*Recursion versus iteration* In subsection 2.3 we discussed the extension of BPA with recursion. In subsection 2.11 we discussed a similar construct: iteration. We can compare both approaches in the following sense:  $\text{BPA}^*$  is less expressive than  $\text{BPAlin}$ .  $\text{BPAlin}$  is  $\text{BPAreC}$  where only finite linear specifications are allowed<sup>1</sup>. In other words,  $\text{BPA}^*$  does not contain non-regular processes. In [Bergstra *et al.*, 1994b] a simple example is given that shows the strictness of the inclusion. Consider the following regular process:

$$\begin{aligned} X &= a \cdot Y + b, \\ Y &= c \cdot X + d. \end{aligned}$$

In figure 7 we give the graph that belongs to this process. This process is not definable in  $\text{BPA}^*$ . In the next theorem we summarize the results. For the proof we refer to [Bergstra *et al.*, 1994b].

**Theorem 2.14.6.**  *$\text{BPA}^*$  is strictly less expressive than  $\text{BPAlin}$ . There is a regular process that cannot be defined in  $\text{BPA}^*$ .*

### 3 Concrete concurrent processes

Up to now, we have discussed the language BPA with many of its extensions. Next, we want to discuss an extension of such significance that we devote a new section to it. It is the notion of parallelism or concurrency that we add

<sup>1</sup>Note that each recursively specifiable process over ACP can also be specified with a possibly infinite number of linear equations. Hence the finiteness constraint.

to our family of basic systems that we treated in the previous section. We will restrict ourselves to concrete concurrency; that is, we do not consider abstraction.

We follow the ACP approach of [Bergstra and Klop, 1984b]. For other approaches to concurrency we refer to Milner's CCS [Milner, 1980; Milner, 1989], [Hennessy, 1988], and Hoare's CSP [Hoare, 1985].

### 3.1 Introduction

In this section, we first extend the BPA family with a parallel construct without interaction; that is, processes can be put in parallel by means of this operator but they cannot communicate with each other. This system is called PA. Then we will extend this theory with extensions that we discussed in the case of BPA (and new extensions). It will turn out that in most cases the extensions can be obtained in the same way as in the BPA case.

Secondly, we extend the parallel construct itself such that communication between parallel processes is also possible, that is, we discuss the system ACP. Then we discuss extensions of ACP, which is in most cases an easy job since they can be obtained in the same way as the extensions for BPA.

Finally, we discuss decidability and expressiveness issues for various systems.

### 3.2 Syntax and semantics of parallel processes

In this subsection we will describe the syntax and semantics of concrete concurrent processes.

#### 3.2.1 The theory PA

We will discuss the equational theory  $PA = (\Sigma_{PA}, E_{PA})$ . This section is based on [Bergstra and Klop, 1982].

The signature  $\Sigma_{PA}$  consists of the signature of BPA plus two binary operators  $\parallel$  and  $\llcorner$ . The operator  $\parallel$  is called (*free*) *merge* or *parallel composition* and the operator  $\llcorner$  is called *left merge*. The left merge was introduced in [Bergstra and Klop, 1982] in order to give a finite axiomatization of the free merge. [Moller, 1989] proved that it is impossible to give a finite axiomatization of the merge without an auxiliary operator.

The set of equations  $E_{PA}$  consists of the equations of BPA in table 1 and the axioms concerning the merge in table 42. We assume in this table that  $a$  ranges over the set of atomic actions. So axioms M2 and M3 are in fact axiom schemes: for each atomic action there are axioms M2 and M3.

We assume that sequential composition binds stronger than both merges and they in turn bind stronger than the alternative composition. So, for instance, the left-hand side of M3 stands for  $(a \cdot x) \llcorner y$  and the brackets in the left-hand side of M4 are necessary.

Table 42. Axioms for the free merge.

$x \parallel y = x \parallel y + y \parallel x$	M1
$a \parallel x = ax$	M2
$ax \parallel y = a(x \parallel y)$	M3
$(x + y) \parallel z = x \parallel z + y \parallel z$	M4

*Intuition* Before we provide the semantics of PA, we give an intuitive meaning to the non-BPA part of PA: the part concerning both merges. We already discussed the BPA part informally in 2.2.1. We recall that we consider the execution of an atomic action to occur at (or to be observed at) a point in time. We start with the signature and then we treat the axioms.

We think of the merge of two processes  $x$  and  $y$  as the process that executes both  $x$  and  $y$  in parallel. We think of the left merge of  $x$  and  $y$  as precisely the same, with the restriction that the first step of the process  $x \parallel y$  comes from its left-hand side  $x$ . We disregard the simultaneous execution of atomic actions here (but see subsection 3.5 where communication comes into play). This leads to the so-called interleaving view, which clarifies the behaviour of the left merge.

This intuition clarifies that axiom M1 is defined in terms of the left merge: the merge of two processes starts either with the left-hand side or with its right-hand side.

The remaining axioms M2–4 define the left merge following the structure of basic terms.

The parallelism in axiom M2 collapses into sequential composition since the first step at the left-hand side is also the last one. After the first step in M3, we obtain full parallelism for the remainders. Axiom M4 simply says that the left merge distributes over the alternative composition. Note that, in general,  $(x + y) \parallel z \neq x \parallel z + y \parallel z$ . So, here we describe an interleaving parallel composition. Also, other forms of parallel composition can be formulated. We already mentioned interleaving extended with simultaneous execution, to be discussed from subsection 3.5 on, but also want to mention so-called *synchronous* parallel composition, by which we can describe clocked systems, where all components proceed in lock-step. A well known process algebra with synchronous parallel composition is SCCS; [Milner, 1983], two references using the present framework are [Bergstra and Klop, 1984b] and [Weijland, 1989].

*Structural induction* We can use structural induction for PA as before for BPA, since basic PA terms are just basic BPA terms. This follows immediately from the theorem to follow (theorem 3.2.4). It states that

parallel composition can be eliminated from closed PA terms.

*Termination* Next, it is our aim to prove that the term rewriting system associated to the equational specification PA is strongly normalizing. In subsection 2.2.2 we already discussed the powerful method of the recursive path ordering. Indeed, we will use this method to prove the desired result but we cannot apply it immediately. We recall that the termination problem more or less reduces to finding the appropriate strict partial ordering on some operators in the signature. The problem that we have with this particular system is that we cannot define a consistent partial ordering on the elements of the signature. First, we will explain this problem and then we will see that a possible solution can be obtained in the same way as for the termination of  $\text{BPA}_\lambda$ ; see section 2.8.1. The problematical pair consists of the rules (RM1, RM3). Analysing this pair we find that if we take the rule RM1 on the one hand, the ordering that does the job is  $\parallel > \ll$ . On the other hand, if we look at RM3, the right choice is the other way around:  $\ll > \parallel$ . This particular problem is tackled by [Bergstra and Klop, 1985]. More detailed information on this problem can be found in a survey on term rewriting that appeared in this series [Klop, 1992, remark 4.11(ii)]. The idea of [Bergstra and Klop, 1985] was to equip the operators  $\parallel$  and  $\ll$  with a rank. Thus yielding a ranked signature for which it is possible to define the desired strict partial ordering. To formalize the ranked signature we first need a notion termed “weight”. Its definition stems from [Bergstra and Klop, 1985]. Note that we already defined this notion in the case of BPA with the state operator; see definition 2.8.1.

**Definition 3.2.1.** Let  $x$  and  $y$  be terms and let  $a$  be an atomic action. The weight of a term  $x$ , notation  $|x|$ , is defined inductively as follows:

- $|a| = 1$
- $|x + y| = \max\{|x|, |y|\}$
- $|x \cdot y| = |x| + |y|$
- $|x \parallel y| = |x| + |y|$
- $|x \ll y| = |x| + |y|$ .

**Table 43.** A term rewriting system for PA.

$(x + y)z \rightarrow xz + yz$	RA4
$(xy)z \rightarrow x(yz)$	RA5
$x \parallel y \rightarrow x \ll y + y \ll x$	RM1
$a \ll x \rightarrow ax$	RM2
$ax \ll y \rightarrow a(x \parallel y)$	RM3
$(x + y) \ll z \rightarrow x \ll z + y \ll z$	RM4



Below we give the definition of a ranked operator as defined by [Bergstra and Klop, 1985]. And we list the new signature.

**Definition 3.2.2.** The rank of an operator  $\parallel$  or  $\ll$  is the weight of the subterm of which it is the leading operator. The signature for the term rewriting system associated with PA is the following:

$$A \cup \{+, \cdot\} \cup \{\parallel_n, \ll_n : n \geq 2\},$$

where the subscripted  $n$  stands for the sum of the weights of the arguments.

Now that we are equipped with the right tools we formulate the termination theorem for the system PA.

**Theorem 3.2.3.** *The term rewriting system associated to PA (see table 43) is strongly normalizing.*

**Proof.** We will give the partial ordering so that we can use the method of the recursive path ordering. We use the following ordering on the signature; this ordering is taken from [Bergstra and Klop, 1985].

$$+ < \cdot < \ll_2 < \parallel_2 < \ll_3 < \parallel_3 < \dots$$

Moreover, we give  $\cdot$  the lexicographical status for the first argument. We will treat RM1 and RM3 to show the use of the ranked operators. First, we display the calculations that lead to the desired inequality concerning RM1. Let  $|x| + |y| = n$ . Notice that we are to show that

$$x \parallel_n y >_{lpo} x \ll_n y + y \ll_n x.$$

We will make use of the fact that  $\parallel_n > +$  and that  $\parallel_n > \ll_n$ .

$$\begin{aligned} x \parallel_n y &>_{lpo} x \parallel_n^* y \\ &>_{lpo} x \parallel_n^* y + x \parallel_n^* y \\ &>_{lpo} (x \parallel_n^* y) \ll_n (x \parallel_n^* y) + (x \parallel_n^* y) \ll_n (x \parallel_n^* y) \\ &>_{lpo} x \ll_n y + y \ll_n x. \end{aligned}$$

Now we handle the case RM3. Let  $|x| + |y| = n$ .

$$\begin{aligned} (a \cdot x) \ll_{n+1} y &>_{lpo} (a \cdot x) \ll_{n+1}^* y \\ &>_{lpo} ((a \cdot x) \ll_{n+1}^* y) \cdot ((a \cdot x) \ll_{n+1}^* y) \\ &>_{lpo} (a \cdot x) \cdot \left( ((a \cdot x) \ll_{n+1}^* y) \parallel_n ((a \cdot x) \ll_{n+1}^* y) \right) \\ &>_{lpo} (a \cdot^* x) \cdot ((a \cdot x) \parallel_n y) \\ &>_{lpo} a \cdot ((a \cdot^* x) \parallel_n y) \end{aligned}$$

$$>_{lpo} a \cdot (x \parallel_n y).$$

The other cases are verified along the same lines. ■

By means of the termination of PA we can now formulate the following elimination theorem, which states that the merge and the left merge can be eliminated for closed terms.

**Theorem 3.2.4.** *For every closed PA term  $t$  there is a basic BPA term  $s$  such that  $PA \vdash t = s$ .*

**Proof.** According to theorem 3.2.3 we find that the term rewriting system of table 43 is strongly normalizing. Let  $t$  be a closed PA term and let  $s$  be its normal form with respect to the term rewriting system of table 43. With proposition 2.2.5 it suffices to show that  $s$  is a closed BPA term. Suppose that  $s$  contains a merge; then we can use RM1, which contradicts the normality of  $s$ . Now suppose that  $s$  contains a left merge and consider the smallest subterm containing it. Due to this minimality, it is of the form  $u \parallel v$  with  $u$  and  $v$  closed BPA terms. Rewrite  $u$  into its BPA normal form. Then RM2, RM3, or RM4 can be applied, which again contradicts the normality of  $s$ . So  $s$  must be a closed BPA term. ■

*Standard concurrency* Some properties concerning both merges cannot be derived from PA, but can only be proved for closed PA terms, for instance the associativity of the merge. In many applications these properties are useful and thus assumed to hold. Hence, following [Bergstra and Tucker, 1984], they are often referred to as *axioms for standard concurrency*. In the next theorem we will treat two such equalities.

**Theorem 3.2.5.** *Let  $x, y$ , and  $z$  be closed PA terms. Then the following two statements hold:*

- (i)  $(x \parallel y) \parallel z = x \parallel (y \parallel z),$
- (ii)  $(x \parallel y) \parallel z = x \parallel (y \parallel z).$

**Proof.** We prove both equalities with induction on the sum  $s$  of the number of symbols occurring in  $x, y$ , and  $z$ . The case  $s = 3$  is trivial so we only treat the case  $s + 1$ . In accordance with theorem 3.2.4, we may assume that  $x$  is a basic BPA term. This gives three trivial cases for the first equality.

To prove the second equality use the fact that the first equality holds for  $s + 1$  and use the (derivable) fact that the merge is commutative. ■

*Expansion* An important result in PA with standard concurrency is the so-called expansion theorem, which is a generalization of axiom M1 (see [Bergstra and Tucker, 1984]). It tells us how the merge of more than two processes can be evaluated. For instance, the merge of three processes  $x, y$ , and  $z$  yields

$$x \parallel (y \parallel z) + y \parallel (x \parallel z) + z \parallel (x \parallel y).$$

**Theorem 3.2.6.** *In PA with standard concurrency we have the following for all open PA terms  $x_1, x_2, \dots, x_n$  and  $n \geq 2$ .*

$$x_1 \parallel x_2 \parallel \dots \parallel x_n = \sum_{i=1}^n x_i \parallel (x_1 \parallel \dots \parallel x_{i-1} \parallel x_{i+1} \parallel \dots \parallel x_n).$$

**Proof.** Straightforward induction on  $n$ . ■

### 3.2.2 Semantics of PA

We will give the semantics of PA by means of a term deduction system  $T(\text{PA})$ . Take for its signature the one of PA and for its rules the ones of BPA in table 5 and the rules concerning the merge in table 44. Bisimulation equivalence is a congruence; see 2.2.31. So the operators of PA can be defined on the quotient of closed PA terms with respect to bisimulation equivalence. The following theorem says that this quotient is a model of PA.

**Theorem 3.2.7.** *The set of closed PA terms modulo bisimulation equivalence is a model of PA.*

**Proof.** A1–A5 are treated as in 2.2.33. M2–M4 are proved as A1. Take for M1 the relation between the left- and right-hand sides of M1, that relates all pairs  $(x' \parallel y', y' \parallel x')$ , and that contains the diagonal. ■

Next, we take care of the conservativity of PA over BPA.

**Theorem 3.2.8.** *The equational specification PA is a conservative extension of the equational specification BPA.*

**Proof.** With the aid of theorem 2.4.15 it is very easy to see that the term deduction system  $T(\text{PA})$  is an operationally conservative extension of the term deduction system  $T(\text{BPA})$  (listed in table 5). With theorem 2.4.19 we also find that this holds up to strong bisimulation equivalence. With the

**Table 44.** Derivation rules of  $T(\text{PA})$ .

$\frac{x \xrightarrow{a} x'}{x \parallel y \xrightarrow{a} x' \parallel y}$	$\frac{y \xrightarrow{a} y'}{x \parallel y \xrightarrow{a} x \parallel y'}$
$\frac{x \xrightarrow{a} \surd}{x \parallel y \xrightarrow{a} y}$	$\frac{y \xrightarrow{a} \surd}{x \parallel y \xrightarrow{a} x}$
$\frac{x \xrightarrow{a} x'}{x \parallel y \xrightarrow{a} x' \parallel y}$	$\frac{x \xrightarrow{a} \surd}{x \parallel y \xrightarrow{a} y}$

above theorem we know that PA is sound with respect to the model induced by  $T(\text{PA})$ , so according to theorem 2.4.24 we find the conservativity. ■

Below we give the completeness theorem for PA.

**Theorem 3.2.9.** *The axiom system PA is a complete axiomatization of the set of closed PA terms modulo bisimulation equivalence.*

**Proof.** With the aid of theorem 2.4.26 and the conservativity of PA over BPA we find the completeness of PA (use also theorems 2.2.35 and 3.2.4). ■

### 3.3 Extensions of PA

In this subsection we will discuss extensions of PA with various features. We already met these extensions when we discussed BPA. We treat the extension of PA with recursion, projections, renaming, the state operator, and iteration. We postpone the extensions of PA with the priority operator and discrete time until we extended the theory PA with deadlock. We deal with  $\text{PA}_\delta$  in subsection 3.3.2. In subsection 3.3.3, we present an application of  $\text{PA}_\delta$  with the state operator, namely in the description of asynchronous communication. We explain in 3.3.4 why we do not treat PA with the empty process.

An extension that is new here is the extension of PA with a process creation mechanism. The reason we did not discuss this extension before, is that this extension makes essential use of the parallel operator. We discuss this extension in subsection 3.3.1.

*Recursion* Here we will add recursion to PA. This is done in the same way as we did for BPA.

The equational specification  $\text{PAREC}$  has as its signature the signature of  $\text{BPAREC}$  plus the two binary operators  $\parallel$  and  $\llbracket$  present in the signature of PA. The axioms of  $\text{PAREC}$  are the ones of  $\text{BPAREC}$  plus the axioms of table 42.

The definition of a guard and a guarded recursive specification are the same as in subsection 2.3. Note that there are more guarded terms and recursion equations (thus guarded recursive specifications) in PA than in BPA. For example,  $a(X \parallel Y)$  is a guarded term and the recursion equation  $X = a \llbracket X$  is guarded because of axiom M2.

The semantics of  $\text{PAREC}$  can be given with a term deduction system  $T(\text{PAREC})$ : take for its signature the one of  $\text{PAREC}$  and for its rules the rules of PA plus the rules concerning recursion; see table 6. Since bisimulation equivalence is a congruence (2.2.31), we can define the operators of  $\text{PAREC}$  on the quotient algebra of the set of closed  $\text{PAREC}$  terms with respect to bisimulation equivalence. This quotient is a model of  $\text{PAREC}$  and it satisfies RDP, AIP<sup>-</sup>, and RSP.

*Projection* We can extend the equational specification PA with projections as we did for BPA. The equational specification  $\text{PA} + \text{PR}$  has as its signature



the one of BPA + PR plus the two binary operators  $\parallel$  and  $\lfloor\rfloor$  present in the signature of PA. Its axioms are the ones of PA plus the axioms concerning projections; see table 7.

The results that we obtained in subsection 2.4 translate effortlessly to the present situation.

*Recursion and projection* Here we will extend PA with both recursion and projection. This extension is obtained analogously to the extension of BPA with recursion and projection. The specification PAREC + PR has as its signature the one of PAREC plus the unary operators  $\pi_n$  that occur in the signature of PA + PR. Its axioms are the ones of PAREC plus the axioms concerning projection; see table 7. The results that we obtained for BPAREC + PR in subsection 2.4.2 also hold for PAREC + PR. We will not mention them here.

The semantics of PAREC + PR can be given by a combination of the term deduction system of PAREC and PA + PR.

*Renaming* It is not difficult to extend the equational specification PA, and its extensions, with renaming operators; see subsections 2.7 and 2.7.1.

*State operator* We can extend the theory PA with either the simple or extended state operator in the same way as we did for the theory BPA. For details we refer to subsections 2.8 and 2.9.

*Iteration* We can extend PA with iteration by just adding the defining axioms for  $*$  in table 37 to the ones for PA. For this theory there is no completeness result present at the time of writing this survey.

### 3.3.1 Process creation

In this subsection, we discuss an extension of PA with an operator that models process creation. This extension is not present in BPA since it is defined using the parallel composition  $\parallel$ . This subsection is based on [Bergstra, 1990]. We refer to [America and Bakker, 1988] and to [Baeten and Vaandrager, 1992] for other approaches to process creation.

We refer to [Bergstra and Klint, 1994] for an application of process creation.

*The theory* The equational specification PA + PCR (process algebra with process creation) is defined in stages. First we take the theory PA where we assume that the set of atomic actions  $A$  contains some special actions: for all  $d$  in some data set  $D$  we assume that  $cr(d) \in A$  and  $\overline{cr}(d) \in A$ . Moreover, we assume the existence of a function, the process creation function,  $\phi$  on  $D$  that assigns to each  $d \in D$  a process term  $\phi(d)$  over PA with the above set of atomic actions  $A$ . Using the function  $\phi$  we add a unary operator  $E_\phi$  to the signature, thus obtaining the signature of PA + PCR. The operator  $E_\phi$  is called the process creation operator.

The equations of PA + PCR are those of PA plus the ones that define  $E_\phi$ .



Table 45. Axioms for the process creation operator.

$E_\phi(a) = a$ , if $a \neq cr(d)$ for $d \in D$	PCR1
$E_\phi(cr(d)) = \overline{cr}(d) \cdot E_\phi(\phi(d))$ , for $d \in D$	PCR2
$E_\phi(a \cdot x) = a \cdot E_\phi(x)$ , if $a \neq cr(d)$ for $d \in D$	PCR3
$E_\phi(cr(d) \cdot x) = \overline{cr}(d) \cdot E_\phi(\phi(d) \parallel x)$ , for $d \in D$	PCR4
$E_\phi(x + y) = E_\phi(x) + E_\phi(y)$	PCR5

We list these equations in table 45.

*Intuition* We provide some intuition for PA + PCR. We will compare process creation with the UNIX<sup>2</sup> system call `fork`; see [Ritchie and Thompson, 1974]. We recall that with `fork` we can only create an exact copy (child process) of its so-called parent process. We note that with the process creation operator we are able to create arbitrary processes but to provide an intuition for process creation the system call `fork` is illustrative.

The atomic action  $cr(d)$  can be seen as a trigger for  $E_\phi$ ; compare  $cr(d)$  to the system call `fork`. The operator  $E_\phi$  initiates the creation of a process when a  $cr(d)$  is parsed; think of it as a program that invokes the system call `fork`. The action  $\overline{cr}(d)$  indicates that a process creation has occurred; this action can be interpreted as the return value of the system call `fork` to the parent process (which is the unique process ID of the newly created process). Maybe this intuition is best illustrated by axiom PCR4. There we see that from  $E_\phi(cr(d) \cdot x)$  a process  $\phi(d)$  is created that is put in parallel with the remaining process  $x$ , while leaving a trace  $\overline{cr}(d)$ .

Next, we formulate a simple lemma that states that process creation distributes over the merge.

**Lemma 3.3.1.** *For all closed PA terms  $x$  and  $y$  we have*

$$E_\phi(x \parallel y) = E_\phi(x) \parallel E_\phi(y).$$

**Proof.** Use structural induction on both  $x$  and  $y$ . ■

**Example 3.3.2.** Let  $D = \{d\}$  and let  $\phi(d) = cr(d)$ . If  $x = E_\phi(cr(d))$ , then  $x = \overline{cr}(d) \cdot x$ . So we see that even the simplest examples give rise to recursive equations.

*Termination* The above example shows that the term rewriting system associated to PA + PCR (by orienting the axioms of PA + PCR from left

---

<sup>2</sup>UNIX is a registered trademark of UNIX System Laboratories (at least at the time of writing this survey).

**Table 46.** Operational rules for the process creation operator.

$\frac{x \xrightarrow{a} x'}{E_\phi(x) \xrightarrow{a} E_\phi(x')}, a \neq cr(d)$	$\frac{x \xrightarrow{a} \surd}{E_\phi(x) \xrightarrow{a} \surd}, a \neq cr(d)$
$\frac{x \xrightarrow{cr(d)} x'}{E_\phi(x) \xrightarrow{\overline{cr}(d)} E_\phi(\phi(d) \parallel x')}, d \in D$	$\frac{x \xrightarrow{cr(d)} \surd}{E_\phi(x) \xrightarrow{\overline{cr}(d)} E_\phi(\phi(d))}, d \in D$

to right) is, in general, not terminating. For, in case of the above example, we have the following infinite sequence of rewritings:

$$E_\phi(cr(d)) \rightarrow \overline{cr}(d) \cdot E_\phi(cr(d)) \rightarrow \overline{cr}(d) \cdot \overline{cr}(d) \cdot E_\phi(cr(d)) \rightarrow \dots$$

*Semantics* We discuss the operational semantics of PA + PCR. It is obtained by means of a term deduction system. The signature is that of PA + PCR; the operational rules are those of PA plus the rules that operationally define the process creation operator  $E_\phi$ . We list them in table 46. The rules of this table originate from [Baeten and Bergstra, 1988b].

The soundness of PA + PCR is easily established.

**Theorem 3.3.3.** *The set of closed PA + PCR terms modulo bisimulation equivalence is a model of PA + PCR.*

**Proof.** For the equations of PA we refer to theorem 3.2.7. So we only need to show the soundness of the equations PCR1–5. This is easy. We only give the bisimulation relations and leave the calculations to the reader. For PCR1, relate the left-hand side and the right-hand side of PCR1. For PCR2–5 also take such a pair and join this with the diagonal. ■

Next, we state that PA + PCR is a conservative extension of PA.

**Theorem 3.3.4.** *The equational specification PA + PCR is a conservative extension of the equational specification PA.*

**Proof.** As usual. ■

From example 3.3.2 it follows that the process creation operator introduces recursion. So we do not have a completeness theorem.

### 3.3.2 Deadlock in PA

It is straightforward to add deadlock to the theory PA. The equational specification  $PA_\delta$  has as its signature the one of PA plus a constant  $\delta \notin A$ . Its axioms are the ones of PA plus the axioms concerning deadlock listed in table 10. We assume for axioms M2 and M3 that  $a$  ranges over the set  $A \cup \{\delta\}$ .

*Structural induction* We can use structural induction for  $\text{PA}_\delta$  as before for  $\text{BPA}_\delta$ , since basic  $\text{PA}_\delta$  terms are just basic  $\text{BPA}_\delta$  terms. This follows immediately from the fact that both merges can be eliminated from closed  $\text{PA}_\delta$  terms. This can be shown by means of a term rewriting analysis just as in theorem 3.2.3 and the following elimination theorem.

**Theorem 3.3.5.** *For every closed  $\text{PA}_\delta$  term  $t$  there is a basic  $\text{BPA}_\delta$  term  $s$  such that  $\text{PA}_\delta \vdash t = s$ .*

**Proof.** This is proved along the same lines as theorem 3.2.4. ■

Also the conservativity of  $\text{PA}_\delta$  over  $\text{BPA}_\delta$  and the completeness of  $\text{PA}_\delta$  can be proved along the same lines as these results for  $\text{PA}$  without extensions.

*Standard concurrency* Standard concurrency in  $\text{PA}_\delta$  is dealt with completely analogously to the situation without deadlock, so we refer to theorem 3.2.5 for standard concurrency. Below, we will mention some properties about the connection of deadlock and parallel composition. The proof of these properties is elementary and therefore omitted.

**Theorem 3.3.6.**

(i)  $\text{PA}_\delta \vdash \delta \parallel x = \delta$ .

Let  $x$  be a closed  $\text{PA}_\delta$  term. Then we have

(ii)  $x \parallel \delta = x \parallel \delta = x\delta$ .

Let  $x$  and  $y$  be closed  $\text{PA}_\delta$  terms. Then we have

(iii)  $x \parallel y\delta = (x \parallel y)\delta = x\delta \parallel y$ .

**Remark 3.3.7.** We mention that if in addition we have standard concurrency, the proof of (iii) follows easily using (ii):

$$x \parallel y\delta = x \parallel (y \parallel \delta) = (x \parallel y) \parallel \delta = (x \parallel y)\delta.$$

*Expansion* For  $\text{PA}_\delta$  with standard concurrency we have the same expansion theorem as for the theory without deadlock, so for expansion we refer to theorem 3.2.6.

*Semantics* The semantics of  $\text{PA}_\delta$  can be given by means of a term deduction system  $T(\text{PA}_\delta)$ , which is just  $T(\text{PA})$  with  $\delta$  added to its signature. The operators of  $\text{PA}_\delta$  can be easily defined by taking representatives on the quotient of the set of closed  $\text{PA}_\delta$  terms modulo bisimulation equivalence, since this relation is a congruence; see 2.2.31. The quotient is a model of  $\text{PA}_\delta$ , which can be easily checked by combining the soundness proofs for  $\text{BPA}_\delta$  and  $\text{PA}$ . Moreover, the axiom system  $\text{PA}_\delta$  is a complete axiomatization of this quotient. This follows immediately from the completeness of  $\text{PA}$  since we did not introduce any new transitions.

### 3.3.3 Asynchronous communication

It is straightforward to extend  $PA_\delta$  with any of the features mentioned in the beginning of subsection 3.3. Here, we consider an application of  $PA_\delta$  with the (simple) state operator. We describe mail through a communication channel. Let  $D$  be a finite data set and let  $c$  be a communication channel. We assume that for each  $d \in D$  we have the following special atomic actions:

- $c \uparrow d$     send  $d$  via  $c$ ; *potential* action
- $c \uparrow\uparrow d$     send  $d$  via  $c$ ; *realized* action
- $c \downarrow d$     receive  $d$  via  $c$ ; *potential* action
- $c \downarrow\downarrow d$     receive  $d$  via  $c$ ; *realized* action.

The state operator will turn potential, intended actions into realized actions. The state space will keep track of outstanding messages.

We consider the case where the communication channel behaves like a queue, i.e. the order of the messages is preserved. Without much trouble, descriptions for other kinds of channels can be generated (for instance, a bag-like channel). Thus, the state space is  $D^*$ , the set of words over  $D$ . Let  $\sigma, \rho$  range over  $D^*$ , and let  $\varepsilon$  denote the empty word. We denote the concatenation of words  $\sigma$  and  $\rho$  simply by  $\sigma\rho$ . Note that  $D \subseteq D^*$  so  $\sigma d$  is the concatenation of the words  $\sigma$  and  $d$  (for  $d \in D$ ). Let  $last(\sigma)$  be the last element of word  $\sigma$ , if  $\sigma \neq \varepsilon$ .

We define the *action* and *effect* functions implicitly, by giving the relevant instances of axiom SO2.

$$\begin{aligned} \lambda_\sigma^c(c \uparrow d \cdot x) &= c \uparrow\uparrow d \cdot \lambda_{d\sigma}^c(x) \\ \lambda_{\sigma d}^c(c \downarrow d \cdot x) &= c \downarrow\downarrow d \cdot \lambda_\sigma^c(x) \\ \lambda_\sigma^c(c \downarrow d \cdot x) &= \delta, & \text{if } \sigma = \varepsilon \text{ or } last(\sigma) \neq d. \end{aligned}$$

The *action* and *effect* functions are inert for all other atomic actions. Now suppose  $0 \in D$ ; then we can describe two communication partners:

$$\begin{aligned} S &= c \uparrow 0, \\ R &= \sum_{d \in D} c \downarrow d \cdot print(d). \end{aligned}$$

Some easy calculations show that

$$\begin{aligned} \lambda_\varepsilon^c(S \parallel R) &= c \uparrow\uparrow 0 \cdot \lambda_0^c(R) = c \uparrow\uparrow 0 \cdot c \downarrow\downarrow 0 \cdot \lambda_\varepsilon^c(print(0)) \\ &= c \uparrow\uparrow 0 \cdot c \downarrow\downarrow 0 \cdot print(0). \end{aligned}$$

Asynchronous communication in the setting of PA was introduced in [Bergstra *et al.*, 1985]. The present formulation is taken from [Baeten and Weijland, 1990].



### 3.3.4 Empty process in PA

We will not discuss the combination of parallel composition and the empty process, since this combination is not (yet) standardized. At this moment there are two possible ways to combine the merge and the empty process. These options originate from the various interpretations of the term  $\varepsilon \parallel x$ . It may seem natural to demand that this equals  $x$ , since  $\varepsilon$  is only capable of terminating successfully, but this perspective leads to a non-associative merge, which is rather unnatural and therefore unwanted [Vrancken, 1986]. The intended interpretation of the left merge is that of the merge with the first action from the left process, so the term  $\varepsilon \parallel x$  cannot proceed, since  $\varepsilon$  cannot perform an action. One of the options is that  $\varepsilon \parallel x$  equals  $\delta$  except if  $x = \varepsilon$ : then it equals  $\varepsilon$ ; see [Vrancken, 1986] for more information. The other option drops this exception and uses a unary operator indicating whether or not a process has a termination option to axiomatize the merge [Baeten and Glabbeek, 1987].

## 3.4 Extensions of $PA_\delta$

In this subsection we will discuss the extensions of  $PA_\delta$  with recursion, projections, renaming, and/or the encapsulation operator, the state operator, the priority operator, iteration, process creation, and discrete time.

*Recursion and/or projection* The extensions of  $PA_\delta$  with recursion, projection, or a combination of both are obtained by simply merging these extensions for  $BPA_\delta$  and PA; see subsections 2.5 and 3.3.

*Renaming and encapsulation* It is straightforward to extend the equational specification  $PA_\delta$ , and its extensions, with renaming operators or the encapsulation operator; cf. subsection 2.7.3.

*State operator* The extension of the theory  $PA_\delta$  with either the simple or extended state operator is obtained in the same way as for the theory PA; see subsections 2.8 and 2.9.

*Priority operator* We can extend the theory  $PA_\delta$  with the priority operator in the same way as the extension of  $BPA_\delta$  with that operator. For details of that extension we refer to section 2.10.

*Iteration* We can extend  $PA_\delta$  with iteration by just adding the defining axioms for  $*$  in table 37 to the ones for  $PA_\delta$ . Only for  $BPA^*$  is the completeness proved at the time of writing this survey.

*Process creation* We can extend  $PA_\delta$  with the process creation operator  $E_\phi$  in the same way as we did for PA. For details we refer to subsection 3.3.1.

### 3.4.1 Discrete time

In this subsection, we extend  $PA_\delta$  with discrete time. With the interaction between the discrete time unit delay  $\sigma_d$  and the leftmerge  $\parallel$  we have to be



**Table 47.** The interaction between the left merge and the discrete time unit delay.

$\sigma_d(x) \parallel \underline{\underline{\delta}} = \underline{\underline{\delta}}$	DTM1
$\sigma_d(x) \parallel (\underline{\underline{a}} + y) = \sigma_d(x) \parallel y$	DTM2
$\sigma_d(x) \parallel (\underline{\underline{a}} \cdot y + z) = \sigma_d(x) \parallel z$	DTM3
$\sigma_d(x) \parallel \sigma_d(y) = \sigma_d(x \parallel y)$	DTM4

a bit careful. We recall that  $x \parallel y$  is  $x$  and  $y$  in parallel but the first action stems from  $x$ . With discrete time present, the question arises if that is possible at all. For instance,  $\sigma_d(\underline{\underline{a}}) \parallel \underline{\underline{b}}$  equals  $\underline{\underline{\delta}}$  in this system as we cannot move to the next time slice in order to let  $a$  happen, since  $b$  must occur in the current time slice. The material of this subsection is based on [Baeten and Bergstra, 1992a].

*The theory* We discuss the equational specification  $\text{PA}_{\delta\text{dt}}$ . Its signature is the one of  $\text{PA}_{\delta}$  (with  $\underline{\underline{a}}$  instead of  $a$  for  $a \in A_{\delta}$ ) plus the discrete time unit delay operator  $\sigma_d$  that we first introduced in  $\text{BPA}_{\text{dt}}$ . The equations of  $\text{PA}_{\delta\text{dt}}$  are the ones of  $\text{BPA}_{\delta\text{dt}}$  plus the equations for the merge that we listed in table 42 (again with  $\underline{\underline{a}}$  instead of  $a$ ) and the equations that represent the interaction between the left merge and the discrete time unit delay; we list the latter axioms in table 47. Incidentally, this axiomatization is new here.

*Termination* Next, we discuss the termination of a term rewriting system associated to the equational specification  $\text{PA}_{\delta\text{dt}}$ . Since we have the left merge in our signature we use the ranked operators that we also used for the termination of  $\text{PA}$ ; cf. subsection 3.2.1.

**Theorem 3.4.1.** *The term rewriting system associated to  $\text{PA}_{\delta\text{dt}}$  consisting of the rewrite rules for  $\text{PA}_{\delta}$ , the rules of table 40, and the equations in table 47 oriented from left to right is strongly normalizing.*

**Proof.** Let us use the theory of subsection 2.2.2. As usual we confine ourselves to giving a partial ordering on the signature. In addition to the ordering that we gave in the termination proof for  $\text{PA}_{\delta}$

$$+ < \cdot < \parallel_2 < \parallel_2 < \parallel_3 < \parallel_3 < \dots,$$

we have the following additional precedence:

$$\sigma_d < +.$$

The rest of the proof consists of straightforward calculations. ■

*Elimination* With the above theorem we can obtain an elimination result for closed terms. However, we cannot obtain this result directly. This is

**Table 48.** The additional rules for the merge and the left merge.

$$\frac{x \xrightarrow{\sigma} x', y \xrightarrow{\sigma} y'}{x \parallel y \xrightarrow{\sigma} x' \parallel y'} \quad \frac{x \xrightarrow{\sigma} x', y \xrightarrow{\sigma} y'}{x \ll y \xrightarrow{\sigma} x' \ll y'}$$

due to the fact that we did not consider a term rewriting analysis modulo the axioms without a clear direction such as A1 and A2. We make the problems a bit more concrete with the following term rewriting *modulo* A1 and A2.

$$\begin{aligned} \sigma_d(\underline{a}) \parallel ((\sigma_d(\underline{b}) + \underline{a}) + \sigma_d(\underline{c})) &= \sigma_d(\underline{a}) \parallel (\underline{a} + (\sigma_d(\underline{b}) + \sigma_d(\underline{c}))) \\ &\rightarrow \sigma_d(\underline{a}) \parallel \sigma_d(\underline{b} + \underline{c}) \\ &\rightarrow \sigma_d(\underline{a} \parallel (\underline{b} + \underline{c})) \\ &\rightarrow \sigma_d(\underline{a} \cdot (\underline{b} + \underline{c})). \end{aligned}$$

So, we see that for the elimination of the left merge we need more than just the termination result above. We will solve this problem in the next theorem.

**Theorem 3.4.2.** *The equational specification  $\text{PA}_{\delta\text{dt}}$  has the elimination property for  $\text{BPA}_{\delta\text{dt}}$ .*

**Proof.** Let  $t$  be a  $\text{PA}_{\delta\text{dt}}$  term. Rewrite  $t$  with the term rewriting system associated with  $\text{PA}_{\delta\text{dt}}$  to a normal form  $t_0$ . It is possible that left merges still occur in the resulting term. Take the minimal subterm of  $t_0$  that contains a left merge  $s \parallel s_1$ . Both  $s$  and  $s_1$  are  $\text{BPA}_{\delta\text{dt}}$  terms. We may assume that  $s$  is of the form  $\sigma_d(s_0)$  (otherwise  $t_0$  would not be in normal form). With the aid of theorem 2.12.3 we know that  $s_1$  can be written in one of the following forms:

- $\sum_{i < n} \underline{a}_i \cdot t_i + \sum_{j < m} \underline{b}_j,$
- $t + \sigma_d(s)$  with  $t$  of the above form.

Now replace  $s_1$  with one of the above forms and rewrite the resulting new term to a normal form and repeat this procedure until all left merges have been eliminated. ■

*Semantics* Now we discuss the operational semantics of  $\text{PA}_{\delta\text{dt}}$ . The semantics of the system  $\text{PA}_{\delta\text{dt}}$  is quite straightforward. In table 48 we list the additional operational rules for the merge and the left merge. The entire semantics of  $\text{PA}_{\delta\text{dt}}$  consists of the one of  $\text{BPA}_{\delta\text{dt}}$  plus the rules in tables 44 and 48.

**Theorem 3.4.3.** *The set of closed  $\text{PA}_{\delta\text{dt}}$  terms modulo bisimulation equivalence is a model of  $\text{PA}_{\delta\text{dt}}$ .*

**Proof.** Since bisimulation equivalence is a congruence, we only need to check the soundness of the axioms of  $\text{PA}_{\delta\text{dt}}$ . We already treated all the axioms except DTM1–4. For these we give the bisimulation relations. For DTM1 relate the left-hand side and the right-hand side. For DTM2–4 also relate the left- and right-hand sides and add the diagonal. ■

**Theorem 3.4.4.** *The equational specification  $\text{PA}_{\delta\text{dt}}$  is a conservative extension of the equational specification  $\text{BPA}_{\delta\text{dt}}$ .*

**Proof.** Easy. ■

Now we have all the prerequisites to state the completeness theorem for  $\text{PA}_{\delta\text{dt}}$ . The proof is as usual and therefore omitted.

**Theorem 3.4.5.** *The axiom system  $\text{PA}_{\delta\text{dt}}$  is a complete axiomatization of the set of closed  $\text{PA}_{\delta\text{dt}}$  terms modulo bisimulation equivalence.*

### 3.5 Syntax and semantics of communicating processes

In this subsection we will extend the meaning of the parallel operator  $\parallel$  that we introduced in subsection 3.2. We will call the ensuing operator  $\parallel$  the merge or parallel composition. We use the name free merge for the merge without communication, that is the merge of PA.

We use the extended merge to model synchronous communication between processes.

#### 3.5.1 The theory ACP

We define the syntax of the equational specification  $\text{ACP} = (\Sigma_{\text{ACP}}, E_{\text{ACP}})$  of [Bergstra and Klop, 1984b].

The signature  $\Sigma_{\text{ACP}}$  consists of the one of  $\text{PA}_{\delta}$  plus a binary operator  $|$ , called the communication merge and the encapsulation operator  $\partial_H$  that we already discussed in subsection 2.7.3 (we recall that  $H \subseteq A$ ). Moreover, we fix a partial function  $\gamma : A \times A \rightarrow A$ , where  $A$  is the set of atomic actions. We call  $\gamma$  the communication function. The communication function is, like  $A$ , a parameter of the theory. It is meant to model the communication between processes. In fact, the communication merge  $|$  is the extension of the communication function to processes. We require that  $\gamma$  is both commutative and associative; that is, if  $\gamma(a, b)$  is defined, it equals  $\gamma(b, a)$  and if  $\gamma(a, \gamma(b, c))$  is defined, it equals  $\gamma(\gamma(a, b), c)$  and vice versa. So, we can leave out the brackets in such formulas and render the latter expression as  $\gamma(a, b, c)$ .

Now we give the set of equations  $\Sigma_{\text{ACP}}$ . This set consists of the equations for  $\text{BPA}_{\delta} + \partial_H$  that we discussed in subsection 2.7.3 and the equations that we list in table 49. See table 21 for the defining axioms of the encapsulation operator. Observe that the equations CM2–4 are the same as the ones that we discussed when we introduced PA. We recall them for the sake of ease.

**Table 49.** Axioms for the merge with communication.

$a \mid b = \gamma(a, b)$ , if $\gamma(a, b)$ defined;	CF1
$a \mid b = \delta$ otherwise.	CF2
$x \parallel y = x \parallel y + y \parallel x + x \mid y$	CM1
$a \parallel x = ax$	CM2
$ax \parallel y = a(x \parallel y)$	CM3
$(x + y) \parallel z = x \parallel z + y \parallel z$	CM4
$(a \cdot x) \mid b = (a \mid b) \cdot x$	CM5
$a \mid (b \cdot x) = (a \mid b) \cdot x$	CM6
$(a \cdot x) \mid (b \cdot y) = (a \mid b) \cdot (x \parallel y)$	CM7
$(x + y) \mid z = x \mid z + y \mid z$	CM8
$x \mid (y + z) = x \mid y + x \mid z$	CM9

Now we discuss the axioms of ACP. The most important one is CM1 where a third possibility for the merge is added. The intended interpretation of this summand  $x \mid y$  is that it is the parallel composition of the two processes  $x$  and  $y$  but that the first step must be a communication. Both processes must be able to perform an action for which  $\gamma$  is defined.

*Terminology* We say that two atomic actions do not communicate if the communication function is not defined for them. We say that an atomic action  $a$  is a communication action if  $a = \gamma(b, c)$  for atomic actions  $b$  and  $c$ . A communication action  $\gamma(b, c)$  is called a binary communication; likewise  $\gamma(a, b, c)$  is called ternary if defined. However, most of the time just using binary communication is enough in applications. See also later on when we discuss so-called handshaking.

*Read/send communication* An important case of binary communication is called read/send communication. The idea is that in the set of atomic actions we have read actions  $r_i(d)$ , send actions  $s_i(d)$ , and communication actions  $c_i(d)$ . The intended meaning of  $r_i(d)$  is to read datum  $d \in D$  at port  $i$ , where the set  $D$  is some finite data set. For  $s_i(d)$  a similar intuition holds. Now  $c_i(d)$  is the result of a communication of  $r_i(d)$  and  $s_i(d)$ : it means transmit the datum  $d$  by communication at port  $i$ . The appropriate communication function is  $\gamma(r_i(d), s_i(d)) = c_i(d)$  and  $\gamma$  is not defined otherwise on  $r_i(d)$ ,  $s_j(d)$ , and  $c_k(d)$  (for ports  $i, j, k$ ). For other atomic actions it is permitted to have some communications defined. The above conventions are due to [Bergstra and Klop, 1986]. An example of the use of read/send communication is given in section 3.6.

*Structural induction* As before we can use structural induction for ACP, since basic ACP terms are just basic  $\text{BPA}_\delta$  terms. This follows from theorem 3.5.5 that states that parallel composition and encapsulation can be eliminated from closed ACP terms.

### 3.5.2 Termination

As before we prove the termination of a term rewriting system associated to ACP (see table 50) with the aid of the theory of subsection 2.2.2. The proof of this fact will more or less be the same as the proof for PA. There we have given some operators a weight to avoid problems with the left merge. Note that in table 50, we rewrite  $a \mid b$  to an atomic action  $c$  or  $\delta$  in order to eliminate the communication merge.

Next, we give the definition of the weight function. It is an extension of definition 3.2.1.

**Definition 3.5.1.** Let  $x$  and  $y$  be terms and let  $a$  be an atomic action. The weight of a term  $x$ , notation  $|x|$ , is defined inductively as follows:

- $|a| = 1$
- $|x + y| = \max\{|x|, |y|\}$
- $|x \cdot y| = |x| + |y|$
- $|x \parallel y| = |x| + |y|$
- $|x \underline{\parallel} y| = |x| + |y|$
- $|x \mid y| = |x| + |y|$
- $|\partial_H(x)| = |x|$ .

Below we give the definition of a ranked operator as defined by [Bergstra and Klop, 1985]. And we list the new signature. This definition is an extension of definition 3.2.2.

**Definition 3.5.2.** The rank of an operator  $\parallel$ ,  $\underline{\parallel}$ , or  $\mid$  is the sum of the weights of its arguments. The signature for the term rewriting system associated with ACP is the following:

$$A \cup \{+, \cdot, \partial_H\} \cup \{\parallel_n, \underline{\parallel}_n, \mid_n : n \geq 2\},$$

where the subscripted  $n$  stands for the weight of the subterm.

**Definition 3.5.3.** Let the term rewriting system associated to ACP consist of the following rules: the term rewriting system associated to  $\text{BPA}_\delta + \partial_H$  and the rules in table 50. For completeness sake, we note that the term rewriting system associated to  $\text{BPA}_\delta + \partial_H$  consists of the rewrite rules of table 2 and the equations in tables 10 and 21 oriented from left to right.

**Theorem 3.5.4.** *The term rewriting system associated to ACP is strongly normalizing.*



**Table 50.** Term rewriting rules for the merge.

$a \mid b \rightarrow c$ , if $\gamma(a, b) = c$ ;	RCF1
$a \mid b \rightarrow \delta$ otherwise.	RCF2
$x \parallel y \rightarrow x \parallel y + y \parallel x + x \mid y$	RCM1
$a \parallel x \rightarrow ax$	RCM2
$ax \parallel y \rightarrow a(x \parallel y)$	RCM3
$(x + y) \parallel z \rightarrow x \parallel z + y \parallel z$	RCM4
$(a \cdot x) \mid b \rightarrow (a \mid b) \cdot x$	RCM5
$a \mid (b \cdot x) \rightarrow (a \mid b) \cdot x$	RCM6
$(a \cdot x) \mid (b \cdot y) \rightarrow (a \mid b) \cdot (x \parallel y)$	RCM7
$(x + y) \mid z \rightarrow x \mid z + y \mid z$	RCM8
$x \mid (y + z) \rightarrow x \mid y + x \mid z$	RCM9

**Proof.** The proof can be given along the same lines as the proof of the termination of the PA system; cf. theorem 3.2.3. We use the partial ordering of the signature in figure 8 and leave the calculations to the reader. ■

With the aid of the above termination result for ACP we can easily prove the following elimination theorem.

**Theorem 3.5.5.** *The equational specification ACP has the elimination property for  $BPA_\delta$ .*

**Proof.** Easy. ■

*Standard concurrency* As for PA we have standard concurrency. That is, there are some properties concerning the merge, left merge, and communication merge that are not derivable for arbitrary open terms, but can be shown to be valid for closed terms (or even for solutions of guarded recursive equations). In many applications these properties are useful and thus these properties are assumed to be valid. This is why these properties are often referred to as axioms for standard concurrency. In the next theorem we list them for ACP. See theorem 3.2.5 for standard concurrency in PA.

**Theorem 3.5.6.** *Let  $x, y$ , and  $z$  be closed ACP terms. Then the following statements hold. They are called axioms of standard concurrency.*

- (i)  $x \mid y = y \mid x$ ,
- (ii)  $x \parallel y = y \parallel x$ ,
- (iii)  $(x \mid y) \mid z = x \mid (y \mid z)$ ,
- (iv)  $(x \parallel y) \parallel z = x \parallel (y \parallel z)$ ,

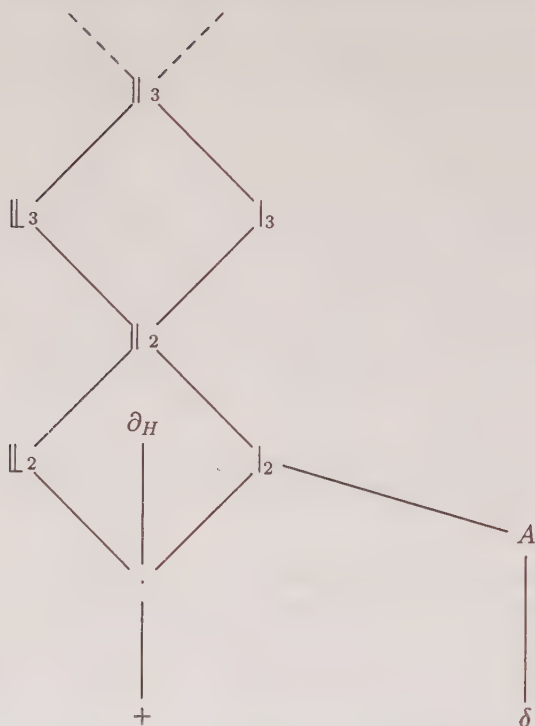


Fig. 8. Partial ordering of the operators in the term rewriting system associated to ACP.

$$(v) \quad x | (y \parallel z) = (x | y) \parallel z,$$

$$(vi) \quad (x \parallel y) \parallel z = x \parallel (y \parallel z).$$

**Proof.** We will not give the details of the proof. They can be found in [Bergstra and Tucker, 1984]. Instead, we explain the proof strategy.

Because of theorem 3.5.5, we only need to prove the properties for basic  $BPA_\delta$  terms. It is easy to show that the first two properties hold by induction to the sum of symbols that occur in both  $x$  and  $y$ . Then it remains to show with a simultaneous induction to the number of symbols occurring in  $x$ ,  $y$ , and  $z$  that the other properties also hold. ■

*Expansion* A useful application of standard concurrency in ACP is the so-called expansion theorem. This theorem states how the merge of more than two processes can be evaluated. For the expansion theorem of PA we refer to theorem 3.2.6. In contrast to the PA expansion theorem, we need an extra proviso for the expansion of the merge in case the communication merge is present. We need a so-called handshaking axiom (HA). We give

**Table 51.** Handshaking axiom.

$x y z = \delta$	HA
------------------	----

it in table 51. It states that there is only binary communication present.

**Theorem 3.5.7.** *Suppose that  $a|b|c = \delta$  for all atomic actions  $a$ ,  $b$ , and  $c$ . Then for all closed ACP terms  $x$ ,  $y$ , and  $z$  we have HA.*

**Proof.** Easy. ■

Next, we formulate the expansion theorem for ACP. The notation that we use in this theorem can be defined inductively in the obvious way.

**Theorem 3.5.8.** *In ACP with standard concurrency and the handshaking axiom presented in table 51 we have the following for all open ACP terms  $x_1, x_2, \dots, x_n$  and  $n \geq 2$ .*

$$x_1 \parallel x_2 \parallel \dots \parallel x_n = \sum_{i=1}^n x_i \parallel \left( \bigsqcup_{\substack{j=1 \\ j \neq i}}^n x_j \right) + \sum_{\substack{i,j=1 \\ i < j}}^n (x_i | x_j) \parallel \left( \bigsqcup_{\substack{k=1 \\ k \neq i,j}}^n x_k \right).$$

**Proof.** Straightforward induction on  $n$ . See [Bergstra and Tucker, 1984] for a detailed proof. ■

### 3.5.3 Semantics of ACP

In this subsection we give the semantics of ACP. In fact, this is now an easy job, since almost all the constructs that ACP contains have been discussed before. The only notion that we did not characterize operationally is the communication merge. In table 52 we present the operational rules for this concept.

The complete operational semantics for ACP is given by a term deduction system  $T(\text{ACP})$  that has as its signature the one of ACP and the rules are the ones of table 5 (the BPA part), the rules of table 22 (the encapsulation part), the rules of table 44 (the PA merge part), and the new rules that we list in table 52. As usual, bisimulation equivalence is a congruence; see 2.2.31. So the operators of ACP can be defined on the quotient of closed ACP terms with respect to bisimulation equivalence. The following theorem says that this quotient is a model of ACP.

**Theorem 3.5.9.** *The set of closed ACP terms modulo bisimulation equivalence is a model of ACP.*

**Proof.** We have already treated the equations that comprise  $\text{BPA}_\delta$ ; see theorem 2.5.4. We have also seen the soundness of  $\text{BPA} + \text{RN}$ , which is BPA with renaming operators. Since the encapsulation operator is a

Table 52. The operational rules for the communication merge.

$\frac{x \xrightarrow{a} x', y \xrightarrow{b} y'}{x \parallel y \xrightarrow{c} x' \parallel y'}, \gamma(a, b) = c$	$\frac{x \xrightarrow{a} x', y \xrightarrow{b} y'}{x \mid y \xrightarrow{c} x' \mid y'}, \gamma(a, b) = c$
$\frac{x \xrightarrow{a} x', y \xrightarrow{b} \surd}{x \parallel y \xrightarrow{c} x'}, \gamma(a, b) = c$	$\frac{x \xrightarrow{a} x', y \xrightarrow{b} \surd}{x \mid y \xrightarrow{c} x'}, \gamma(a, b) = c$
$\frac{x \xrightarrow{a} \surd, y \xrightarrow{b} y'}{x \parallel y \xrightarrow{c} y'}, \gamma(a, b) = c$	$\frac{x \xrightarrow{a} \surd, y \xrightarrow{b} y'}{x \mid y \xrightarrow{c} y'}, \gamma(a, b) = c$
$\frac{x \xrightarrow{a} \surd, y \xrightarrow{b} \surd}{x \parallel y \xrightarrow{c} \surd}, \gamma(a, b) = c$	$\frac{x \xrightarrow{a} \surd, y \xrightarrow{b} \surd}{x \mid y \xrightarrow{c} \surd}, \gamma(a, b) = c$

special case of a renaming operator, the soundness of the equations D1–4 can be proved in the same way as the soundness proof of BPA + RN; see theorem 2.7.4. We have also seen the soundness of the equations CM2–4, since these axioms are the same in PA; see theorem 3.2.7. So it remains to prove that the other equations are sound. We confine ourselves to only giving the bisimulation relations. We begin with CF1. Take the relation that relates both sides of CF1. Equation CF2 is treated exactly the same. Now we treat CM1. Take for CM1 the relation that relates both sides of CM1, that relates  $x' \parallel y'$  and  $y' \parallel x'$  for all closed ACP terms  $x'$  and  $y'$ , and that contains the diagonal. We recall that CM2–4 are treated the same as M2–4 of PA. So we continue with CM5–8. These are proved analogously to A1; that is, relate both sides of an equation and add the diagonal. This ends the soundness proof for ACP. ■

At this point, we are able to prove the conservativity of ACP over  $\text{BPA}_\delta$ .

**Theorem 3.5.10.** *The equational specification ACP is a conservative extension of the equational specification  $\text{BPA}_\delta$ .*

**Proof.** With the aid of theorem 2.4.15 it is very easy to see that the term deduction system  $T(\text{ACP})$  is an operationally conservative extension of the term deduction system  $T(\text{BPA}_\delta)$  (for the definition of  $T(\text{BPA}_\delta)$  we refer to the soundness theorem 2.5.4 for  $\text{BPA}_\delta$ ). With theorem 2.4.19 we also find that this holds up to strong bisimulation equivalence. With theorem 3.5.9 we know that ACP is sound with respect to the model induced by  $T(\text{ACP})$ , so according to theorem 2.4.24 we immediately find the equational conservativity. ■

Below we give the completeness theorem for ACP.

**Theorem 3.5.11.** *The axiom system ACP is a complete axiomatization of the set of closed ACP terms modulo bisimulation equivalence.*

**Proof.** With the aid of theorem 2.4.26 and the conservativity of ACP over  $\text{BPA}_\delta$  we find the completeness of ACP (use also theorems 2.5.5 and 3.5.5). ■

### 3.6 Extensions of ACP

In this subsection we will discuss extensions of ACP with the features that we already met when we discussed extensions of both BPA and PA. We treat the extension of ACP with recursion, projections, renaming operators, the state operator, the priority operator, iteration, process creation, and discrete time. We will also treat two examples to illustrate the use of two of the extensions. We do not discuss the extension of ACP with the empty process. We explained why in subsection 3.3.4.

*Recursion and/or projection* The extensions of ACP with recursion, projection, or a combination of both are obtained by simply merging these extensions for  $\text{BPA}_\delta$  and ACP; see subsection 2.5.

**Example 3.6.1.** In  $\text{ACPrec}$ , we can describe communicating buffers using the read/send communication function defined in subsection 3.5.1. Let  $D$  be a finite data set. A *one-place buffer* over  $D$ , with input port 1 and output port 2, is given by the recursive equation

$$B = \sum_{d \in D} r_1(d) \cdot s_2(d) \cdot B.$$

Likewise, a one-place buffer with input port 2 and output port 3 is given by

$$C = \sum_{d \in D} r_2(d) \cdot s_3(d) \cdot C.$$

Now if  $H = \{r_2(d), s_2(d) : d \in D\}$ , then expression

$$\partial_H(B \parallel C)$$

describes a system of two communicating buffers. Some calculations, and using RSP, show that this system is a solution of the following recursive specification:

$$\begin{aligned} X &= \sum_{d \in D} r_1(d) \cdot c_2(d) \cdot X_d, \\ X_d &= s_3(d) \cdot X + \sum_{e \in D} r_1(e) \cdot s_3(d) \cdot c_2(e) \cdot X_e. \end{aligned}$$

This definition can be seen as defining a *two-place buffer*.



*Renaming and encapsulation* It is straightforward to extend the equational specification ACP, and its extensions, with renaming operators; cf. subsection 2.7.3.

*State operator* The extension of the theory ACP with either the simple or extended state operator is obtained in the same way as for the theories BPA or PA; see subsections 2.8 and 2.9.

*Priority operator* We can extend the theory ACP with the priority operator in the same way as we extended BPA<sub>δ</sub> with that operator. For the details we refer to subsection 2.10.

In the system ACP<sub>θ</sub>, we can describe forms of *asymmetric communication*. Notice that ACP itself features symmetric communication: both ‘halves’ of a communication action must be present before either one can proceed.

**Example 3.6.2.** A *put* mechanism describes a sending action that does not wait for a corresponding receiving action; the message is lost if it cannot be received. If a receiving action is present, then communication should occur.

For a port  $i$  and message  $d$ , we have actions  $put_i(d)$ ,  $r_i(d)$ ,  $c_i(d)$  with the communication function given by

$$\gamma(put_i(d), r_i(d)) = c_i(d) \quad \gamma \text{ not defined otherwise.}$$

The priority ordering is given by  $put_i(d) < c_i(d)$  for all  $d$ . Then, if  $S$  has actions  $put_i(d)$ , and  $R$  actions  $r_i(d)$ , put communication is described by the expression

$$\partial_H \circ \theta(S \parallel R),$$

where  $H = \{r_i(d) : d \in D\}$ .

Similarly, we can describe a *get* mechanism, where a process tries to receive a message. When no message is available, an error message  $\perp$  will be read. We have actions  $get_i(d)$ ,  $s_i(d)$ ,  $c_i(d)$  ( $d \in D$ ), and an action  $get_i(\perp)$ . Communication is given by

$$\gamma(s_i(d), get_i(d)) = c_i(d) \quad \gamma \text{ not defined otherwise,}$$

and a priority ordering  $get_i(\perp) < c_i(d)$  for all  $d$ . The system is described as

$$\partial_H \circ \theta(S \parallel R),$$

where  $H = \{get_i(d), s_i(d) : d \in D\}$ , and  $R$  typically has the form

$$R = \sum_{d \in D} get_i(d) \cdot X_d + get_i(\perp) \cdot X_{\perp}.$$

**Table 53.** An extra axiom for Kleene's binary star operator.

$\partial_H(x^*y) = \partial_H(x)^* \partial_H(y)$	BKS4
----------------------------------------------------	------

This material on put and get communication is based on [Bergstra, 1985], more information can also be found in [Baeten and Weijland, 1990].

*Iteration* We can extend ACP with iteration by adding the defining axioms for  $*$  in table 37 to the ones for ACP and one more axiom that gives the relation between Kleene's star and the encapsulation operator. We give this axiom in table 53. We denote this system by  $ACP^*$ . Only for  $BPA^*$  the completeness is proved at the time of writing this survey.

*Process creation* We discuss the extension of ACP with the process creation operator  $E_\phi$  (for the basic definitions we refer to subsection 3.3.1). We recall that the process creation operator is defined in terms of the parallel composition. So it will not be surprising that we need to impose restrictions on the communication behaviour of the special atomic actions  $cr(d)$ . The restrictions are that  $cr(d)$  does not communicate and is not the result of any communication ( $cr(d)$  is not a communication action). In a formula:  $cr(d) \mid a = \delta$  and for all  $a, b \in A : a \mid b \neq cr(d)$ . We note that lemma 3.3.1 only holds when the communication function is trivial; that is, for all  $a, b \in A : \gamma(a, b)$  is undefined.

Furthermore, the only difference with the discussion in subsection 3.3.1 is the restrictions on the  $cr(d)$ -actions with respect to the communication. So for more details we refer to that subsection.

*Inconsistent combinations* If we combine *unguarded* recursion with  $ACP_\theta$  we will find an inconsistency. We show this with an example that originates from [Baeten and Bergstra, 1988b]. Suppose that there are three atomic actions  $r$ ,  $s$ , and  $c$  and  $r \mid s = c$ . Let  $c > s$  be the partial ordering. Now consider the following recursion equation:

$$X = r \parallel (\theta \circ \partial_{\{c\}}(s + X)).$$

With the operational rules we can infer that  $X \xrightarrow{c} \surd \iff X \not\xrightarrow{c} \surd$ .

### 3.6.1 Discrete time

In this subsection we add relative discrete time to ACP. In subsection 3.4.1, we already extended  $PA_\delta$  with this feature. So we only need to clarify the interaction between the discrete time unit delay  $\sigma_d$  and the communication merge and the relation between  $\sigma_d$  and the encapsulation operator.

*The theory* The equational specification  $ACP_{dt}$  consists of the signature that is the union of the signatures of ACP (with  $\underline{a}$  instead of  $a$  for  $a \in A_\delta$ )

**Table 54.** The interaction between  $\sigma_d$  and communication and encapsulation.

$a \mid \sigma_d(x) = \delta$	DTM5
$\sigma_d(x) \mid a = \delta$	DTM6
$(a \cdot x) \mid \sigma_d(y) = \delta$	DTM7
$\sigma_d(x) \mid (a \cdot y) = \delta$	DTM8
$\sigma_d(x) \mid \sigma_d(y) = \sigma_d(x \mid y)$	DTM9
$\partial_H(\sigma_d(x)) = \sigma_d(\partial_H(x))$	DTD

and  $\text{PA}_{\delta\text{dt}}$ . The equations of this specification are the ones of ACP (again with  $\underline{a}$  instead of  $a$ ) plus the rules of table 47 (they represent the interaction between  $\sigma_d$  and the left merge) and some new axioms that we present in table 54; they express the interaction between the discrete time unit delay operator and the communication merge and the interaction between  $\sigma_d$  and the encapsulation operator.

*Termination* The termination of a term rewriting system associated to  $\text{ACP}_{\text{dt}}$  can be obtained with the aid of the method that we discussed in subsection 2.2.2. We can prove the termination by merging the termination proofs of ACP and  $\text{PA}_{\delta\text{dt}}$  and a small addition.

**Theorem 3.6.3.** *The term rewriting system consisting of the rewrite rules for ACP (see definition 3.5.3) plus the axioms listed in tables 47 and 54 oriented from left to right is strongly normalizing (or terminating).*

**Proof.** The partial ordering on the signature is as follows. Take the one for ACP; see figure 8. Now add the following to this partial ordering:

- $+ > \sigma_d$ ,
- $\sigma_d > \underline{\delta}$ ,
- $\sigma_d > \partial_H$ .

With the proofs of theorems 3.5.4 and 3.4.1 we find that the term rewriting system associated to  $\text{ACP}_{\text{dt}}$  is strongly normalizing. We omit these inferences as they are straightforward. ■

With the above theorem it is easy to obtain an elimination result for closed terms.

**Theorem 3.6.4.** *The equational specification  $\text{ACP}_{\text{dt}}$  has the elimination property for  $\text{BPA}_{\delta\text{dt}}$*

**Proof.** Easy. ■

*Semantics* The semantics of  $\text{ACP}_{\text{dt}}$  can be given with a term deduction system  $T(\text{ACP}_{\text{dt}})$ . Its signature is that of  $\text{ACP}_{\text{dt}}$ . The rules are those of

**Table 55.** The additional rules for the communication merge and the encapsulation operator.

$\frac{x \xrightarrow{\sigma} x', y \xrightarrow{\sigma} y'}{x \mid y \xrightarrow{\sigma} x' \mid y'}$	$\frac{x \xrightarrow{\sigma} x'}{\partial_H(x) \xrightarrow{\sigma} \partial_H(x')}$
---------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------

ACP (with  $\underline{a}$  instead of  $a$  for  $a \in A_\delta$ ), those in table 48 (they concern the merge and the left merge), and the operational rules that we present in table 55. The two rules in table 55 define what kind of  $\sigma$  transitions the communication merge and the encapsulation operator can perform.

**Theorem 3.6.5.** *The set of closed  $\text{ACP}_{\text{dt}}$  terms modulo strong bisimulation equivalence is a model of  $\text{ACP}_{\text{dt}}$ .*

**Proof.** Bisimulation equivalence is a congruence, so we only need to check the soundness of the axioms. The only axioms that we have not checked yet in other soundness proofs are the ones of table 54. They are all very easy. For DTM5–8 we only need to relate the left- and right-hand side. For DTM9 and DTD we additionally include the diagonal. ■

**Theorem 3.6.6.** *The equational specification  $\text{ACP}_{\text{dt}}$  is a conservative extension of the equational specification  $\text{BPA}_{\delta\text{dt}}$ .*

**Proof.** Easy. ■

Now we have all the prerequisites to state the completeness theorem for  $\text{ACP}_{\text{dt}}$ . The proof is as usual and therefore omitted.

**Theorem 3.6.7.** *The axiom system  $\text{ACP}_{\text{dt}}$  is a complete axiomatization of the set of closed  $\text{ACP}_{\text{dt}}$  terms modulo bisimulation equivalence.*

### 3.7 Decidability and expressiveness results in ACP

In subsection 2.14 we discussed the decidability of bisimulation equivalence for  $\text{BParec}$  and we showed that  $\text{BParec}$  can express non-regular processes. In this subsection we will briefly discuss decidability and expressiveness results for the  $\text{Parec}$  and  $\text{ACPrec}$  families.

#### 3.7.1 Decidability

At the time of writing this survey the results are that bisimulation equivalence is undecidable for  $\text{ACPrec}$  and the problem is open for  $\text{Parec}$ . However, some results have been obtained in the direction of  $\text{Parec}$ . For the so-called Basic Parallel Processes (BPP) [Christensen *et al.*, 1993] the problem is solved: BPP is decidable. The equational theory of BPP is close to  $\text{PA}_{\delta\text{rec}}$  with prefix sequential composition instead of sequential composition.

We will formulate these results below.

The next theorem is due to [Bergstra and Klop, 1984b]. For the proof of this fact we refer to [Bergstra and Klop, 1984b].

**Theorem 3.7.1.** *The bisimulation equivalence problem for finitely recursively defined processes over ACP is undecidable.*

For the decidability result on basic parallel processes we briefly introduce the syntax and semantics of this system.

*Basic parallel processes* We will introduce the syntax and semantics of BPP below using the notation that we are used to in this survey. We have a special constant  $\delta$ , alternative composition  $+$ , parallel composition  $\parallel$ , and a unary prefix operator  $a_-$ , called prefix sequential composition, for all  $a \in A$ , where  $A$  is some set. Now if we also add recursion we have the syntax of BPP.

The semantics of BPP is given by means of the term deduction system in table 56. For all the operators we have the usual operational rules but only the non-predicate parts. We have not seen the well-known operational characterization of prefix sequential composition before in this chapter. We give the rules for BPP in one table for the sake of ease.

We note that bisimulation equivalence in this case is just the one that we defined in definition 2.2.27 without the predicate part.

The next theorem is taken from [Christensen *et al.*, 1993]. For the proof of this fact we refer to [Christensen *et al.*, 1993].

**Theorem 3.7.2.** *Bisimulation equivalence is decidable for basic parallel processes (BPP).*

### 3.7.2 Expressiveness

In this subsection we discuss various expressivity results. It turns out that ACPrec is more expressive than PAreC and that the latter is more expressive than BPArec.

*The bag* We consider a so-called bag of unbounded capacity. A bag is a process able to input data elements that reappear in some arbitrary order. Let  $D$  be a finite set of such datum elements containing more than one

**Table 56.** The operational semantics of BPP.

$ax \xrightarrow{a} x$	$\frac{\langle s_X   E \rangle \xrightarrow{a} y}{\langle X   E \rangle \xrightarrow{a} y}$
$\frac{x \xrightarrow{a} x'}{x + y \xrightarrow{a} x'}$	$\frac{y \xrightarrow{a} y'}{x + y \xrightarrow{a} y'}$
$\frac{x \xrightarrow{a} x'}{x \parallel y \xrightarrow{a} x' \parallel y}$	$\frac{y \xrightarrow{a} y'}{x \parallel y \xrightarrow{a} x \parallel y'}$



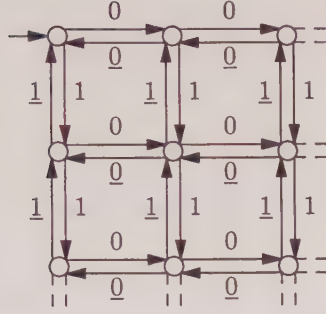


Fig. 9. A deduction graph of a bag over two datum elements.

datum. Suppose that we have atomic actions for all  $d \in D$ :

- $r_1(d)$  means put a  $d$  in the bag;
- $s_2(d)$  means remove a  $d$  from the bag.

The following recursive equation formally defines the bag.

$$B = \sum_{d \in D} r_1(d) \cdot (B \parallel s_2(d)).$$

It will be clear that  $B$  is definable over PArec. In figure 9 we depict the deduction graph of a bag over two datum elements 0 and 1. Note that we abbreviate  $r_1(d)$  to  $d$  and  $s_2(d)$  to  $\underline{d}$  for  $d = 0, 1$ .

Next, we state that PArec is more expressive than BParec. This theorem stems from [Bergstra and Klop, 1984a; Bergstra and Klop, 1995]. For its proof we refer to this paper.

**Theorem 3.7.3.** *A bag over more than one datum element cannot be given by means of a finite recursive specification in BParec. So, PArec is more expressive than BParec.*

**Remark 3.7.4.** Observe that the bag can also be specified in BPP. But since there is a process (the stack) that can be defined in BParec and *not* in BPP the systems are incomparable as far as expressivity is concerned. See [Christensen, 1993] for more details.

Next, we consider the expressivity of ACPrec over PArec. The following theorem is taken from [Bergstra and Klop, 1984a; Bergstra and Klop, 1995]. For more details on this result and its proof we refer to this paper.

**Theorem 3.7.5.** *The process  $p = ba(ba^2)^2(ba^3)^2(ba^4)^2 \dots$  cannot be defined in PArec with a set of atomic actions  $\{a, b\}$  but  $p$  can be defined in ACPrec with atomic actions  $\{a, b, c, d\}$  and with communication function  $\gamma(c, c) = a$ ,  $\gamma(d, d) = b$  (other communications yield  $\delta$ ).*



Fig. 10. A FIFO queue.

Next, we discuss a result that states that  $\text{ACPrec} + \text{RN}$  is more expressive than  $\text{ACPrec}$ .

*The queue* A queue is a process that transmits incoming data while preserving their order. See also figure 10. Such a process is also called a FIFO (First In First Out) queue. First, we describe the queue with input port 1 and output port 2 over a finite data set  $D$  by means of an infinite linear specification. As in 3.3.3,  $D^*$  is the set of words over  $D$ .

It is not hard to see that a queue with input port 1 and output port 2 over the data set  $D$  can be specified as follows:

$$\begin{aligned} Q &= Q_\varepsilon = \sum_{d \in D} r_1(d) \cdot Q_d, \\ Q_{\sigma d} &= s_2(d) \cdot Q_\sigma + \sum_{e \in D} r_1(e) \cdot Q_{e\sigma d}. \end{aligned}$$

We have the last equation for all  $\sigma \in D^*$  and  $d \in D$ .

In figure 11 we give a deduction graph of a queue over two datum elements; note that we abbreviate  $r_1(d)$  by  $d$  and  $s_2(d)$  by  $\underline{d}$  for  $d = 0, 1$ .

The next theorem states that there is no finite specification for the queue in  $\text{ACPrec}$ . This result is taken from [Baeten and Bergstra, 1988a]; the proof uses results from [Bergstra and Tiuryn, 1987]. For a proof we refer to [Baeten and Bergstra, 1988a].

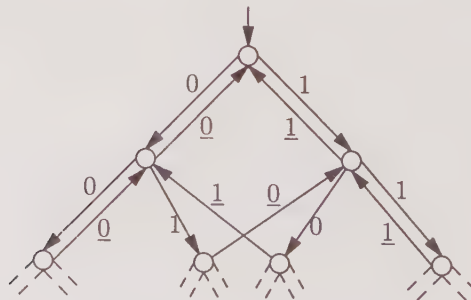


Fig. 11. A deduction graph for the queue.

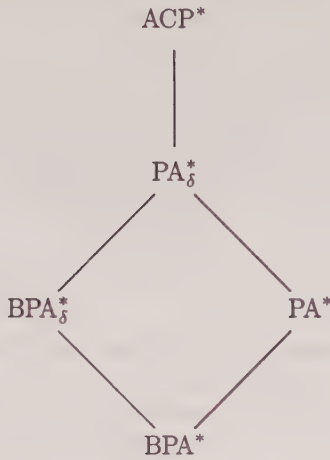


Fig. 12. Expressivity results for systems with iteration.

**Theorem 3.7.6.** *The queue is not finitely definable over  $\text{ACP}_{\text{Prec}}$  using the usual read/send communication that we discussed in subsection 3.5.1.*

In [Baeten and Bergstra, 1988a] it is shown that in  $\text{ACP}_{\text{Prec}} + \text{RN}$  there exists a finite specification of the queue. For details we refer to [Baeten and Bergstra, 1988a].

**Theorem 3.7.7.** *The queue is finitely definable over  $\text{ACP}_{\text{Prec}} + \text{RN}$  using the usual read/send communication that we discussed in subsection 3.5.1.*

Next, we list some expressivity results for extensions of PA and ACP with iteration. These results are taken from [Bergstra *et al.*, 1994b].

**Theorem 3.7.8.** *If there are at least six atomic actions we have the expressivity results for the systems  $\text{BPA}^*$ ,  $\text{BPA}_{\delta}^*$ ,  $\text{PA}^*$ ,  $\text{PA}_{\delta}^*$ , and  $\text{ACP}^*$  as in figure 12. The systems  $\text{BPA}_{\delta}^*$  and  $\text{PA}^*$  are incomparable and for the other systems we have that a line down to a system indicates that the upper system is more expressive than the lower one.*

*Recursion versus iteration* In subsection 2.14 we devoted a small paragraph to the comparison of recursion as treated in subsection 2.3 and iteration (see subsection 2.11). We stated that the BPAlin system (BPA with finite linear recursion) is more expressive than  $\text{BPA}^*$  (see theorem 2.14.6). This result is in fact stronger: the regular system of figure 7 cannot be expressed in  $\text{ACP}^*$ .

In [Bergstra *et al.*, 1994b] it is shown that the regular process depicted in figure 7 can be expressed in  $\text{ACP}^*$  with abstraction.

The next theorem is taken from [Bergstra *et al.*, 1994b]. For the proof we refer to their paper.

**Theorem 3.7.9.** *Not every regular process can be expressed in ACP\* (not even using auxiliary actions).*

More information on the subject of expressiveness in ACP can be found in [Baeten *et al.*, 1987] and in [Vaandrager, 1993]. For more information on expressiveness in systems related to ACP we refer to [Ponse, 1992] or [Glabbeek, 1995].

## 4 Further reading

For those readers who want to know more about process algebra, we give some references. First of all, we want to mention a couple of textbooks in the area. A textbook for CCS-style process algebra is [Milner, 1989], for CSP style we refer to [Hoare, 1985], and for testing theory, there is [Hennessy, 1988]. In ACP style, the standard reference is [Baeten and Weijland, 1990]. The companion volume [Baeten, 1990] discusses applications of this theory. We also want to mention the proceedings of a workshop on ACP style process algebra [Ponse *et al.*, 1995].

When process algebra is applied to larger examples, the need arises to handle data structures also in a formal way. The combination of processes and data is treated in the theories LOTOS [Brinksma, 1987], PSF [Mauw and Veltink, 1993], or  $\mu$ CRL [Groote and Ponse, 1994b].

Tool support in the use of process algebra is provided by most systems; a few references are [Boudol *et al.*, 1990], [Cleaveland *et al.*, 1990], [Godskesen *et al.*, 1989], [Lin, 1992], and [Veltink, 1993].

For an impression of the state of the art in concurrency theory we refer to the proceedings of the series of CONCUR conferences on concurrency theory: [Baeten and Klop, 1990], [Baeten and Groote, 1991], [Cleaveland, 1992], [Best, 1993], and [Jonsson and Parrow, 1994].

## References

- [Aceto and Hennessy, 1992] L. Aceto and M. Hennessy. Termination, deadlock, and divergence. *Journal of the ACM*, 39(1):147–187, January 1992.
- [Aceto *et al.*, 1994] L. Aceto, B. Bloom, and F.W. Vaandrager. Turning SOS rules into equations. *Information and Computation*, 111(1):1–52, 1994.
- [America and Bakker, 1988] P. America and J.W. de Bakker. Designing equivalent semantic models for process creation. *Theoretical Computer Science*, 60:109–176, 1988.
- [Austry and Boudol, 1984] D. Austry and G. Boudol. Algèbre de processus et synchronisations. *Theoretical Computer Science*, 30(1):91–131, 1984.

- [Baeten, 1990] J.C.M. Baeten, editor. *Applications of Process Algebra*. Cambridge Tracts in Theoretical Computer Science 17. Cambridge University Press, 1990.
- [Baeten and Bergstra, 1988a] J.C.M. Baeten and J.A. Bergstra. Global renaming operators in concrete process algebra. *Information and Computation*, 78(3):205–245, 1988.
- [Baeten and Bergstra, 1988b] J.C.M. Baeten and J.A. Bergstra. Processen en procesexpressies. *Informatie*, 30(3):214–222, 1988. In Dutch.
- [Baeten and Bergstra, 1991] J.C.M. Baeten and J.A. Bergstra. Recursive process definitions with the state operator. *Theoretical Computer Science*, 82:285–302, 1991.
- [Baeten and Bergstra, 1992a] J.C.M. Baeten and J.A. Bergstra. Discrete time process algebra (extended abstract). In Cleaveland [1992], pages 401–420. Full version, report P9208b, Programming Research Group, University of Amsterdam, 1992.
- [Baeten and Bergstra, 1992b] J.C.M. Baeten and J.A. Bergstra. Process algebra with signals and conditions. In M. Broy, editor, *Programming and Mathematical Methods, Proceedings Summer School Marktoberdorf 1991*, pages 273–323. Springer-Verlag, 1992. NATO ASI Series F88.
- [Baeten and Bergstra, 1993] J.C.M. Baeten and J.A. Bergstra. Real space process algebra. *Formal Aspects of Computing*, 5(6):481–529, 1993.
- [Baeten and Glabbeek, 1987] J.C.M. Baeten and R.J. van Glabbeek. Merge and termination in process algebra. In K.V. Nori, editor, *Proceedings 7<sup>th</sup> Conference on Foundations of Software Technology and Theoretical Computer Science*, Pune, India, volume 287 of *Lecture Notes in Computer Science*, pages 153–172. Springer-Verlag, 1987.
- [Baeten and Groote, 1991] J.C.M. Baeten and J.F. Groote, editors. *Proceedings CONCUR 91*, Amsterdam, volume 527 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.
- [Baeten and Klop, 1990] J.C.M. Baeten and J.W. Klop, editors. *Proceedings CONCUR 90*, Amsterdam, volume 458 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.
- [Baeten and Vaandrager, 1992] J.C.M. Baeten and F.W. Vaandrager. An algebra for process creation. *Acta Informatica*, 29(4):303–334, 1992.
- [Baeten and Verhoef, 1993] J.C.M. Baeten and C. Verhoef. A congruence theorem for structured operational semantics with predicates. In Best [1993], pages 477–492.
- [Baeten and Weijland, 1990] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.
- [Baeten et al., 1986] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop. Syntax and defining equations for an interrupt mechanism in process algebra.



*Fundamenta Informaticae*, IX(2):127–168, 1986.

- [Baeten *et al.*, 1987] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop. On the consistency of Koomen's fair abstraction rule. *Theoretical Computer Science*, 51(1/2):129–176, 1987.
- [Baeten *et al.*, 1991] J.C.M. Baeten, J.A. Bergstra, S. Mauw, and G.J. Veltink. A process specification formalism based on static COLD. In J.A. Bergstra and L.M.G. Feijs, editors, *Algebraic Methods II: Theory, Tools and Applications*, volume 490 of *Lecture Notes in Computer Science*, pages 303–335. Springer-Verlag, 1991.
- [Baeten *et al.*, 1992] J.C.M. Baeten, J.A. Bergstra, and S.A. Smolka. Axiomatizing probabilistic processes: ACP with generative probabilities. In Cleaveland [1992], pages 472–485.
- [Baeten *et al.*, 1993] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop. Decidability of bisimulation equivalence for processes generating context-free languages. *Journal of the ACM*, 40(3):653–682, July 1993.
- [Bakker and Zucker, 1982] J.W. de Bakker and J.I. Zucker. Processes and the denotational semantics of concurrency. *Information and Control*, 54(1/2):70–120, 1982.
- [Bergstra, 1985] J.A. Bergstra. Put and get, primitives for synchronous unreliable message passing. Logic Group Preprint Series Nr. 3, CIF, State University of Utrecht, 1985.
- [Bergstra, 1990] J.A. Bergstra. A process creation mechanism in process algebra. In Baeten [1990], pages 81–88.
- [Bergstra and Klint, 1994] J.A. Bergstra and P. Klint. The TOOLBUS—a component interconnection architecture. Report P9408, Programming Research Group, University of Amsterdam, 1994.
- [Bergstra and Klop, 1982] J.A. Bergstra and J.W. Klop. Fixed point semantics in process algebras. Report IW 206, Mathematisch Centrum, Amsterdam, 1982.
- [Bergstra and Klop, 1984a] J.A. Bergstra and J.W. Klop. The algebra of recursively defined processes and the algebra of regular processes. In J. Paredaens, editor, *Proceedings 11<sup>th</sup> ICALP, Antwerpen*, volume 172 of *Lecture Notes in Computer Science*, pages 82–95. Springer-Verlag, 1984. Extended abstract, full version appeared in [Ponse *et al.*, 1995].
- [Bergstra and Klop, 1984b] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1/3):109–137, 1984.
- [Bergstra and Klop, 1985] J.A. Bergstra and J.W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37(1):77–121, 1985.
- [Bergstra and Klop, 1986] J.A. Bergstra and J.W. Klop. Verification of an alternating bit protocol by means of process algebra. In W. Bibel and

- K.P. Jantke, editors, *Math. Methods of Spec. and Synthesis of Software Systems '85*, *Math. Research* 31, pages 9–23, Berlin, 1986. Akademie-Verlag.
- [Bergstra and Klop, 1995] J.A. Bergstra and J.W. Klop. The algebra of recursively defined processes and the algebra of regular processes. In Ponse et al. [1995], pages 1–25. Extended abstract of [Bergstra and Klop, 1984a].
- [Bergstra and Tiuryn, 1987] J.A. Bergstra and J. Tiuryn. Process algebra semantics for queues. *Fundamenta Informaticae*, X:213–224, 1987.
- [Bergstra and Tucker, 1984] J.A. Bergstra and J.V. Tucker. Top down design and the algebra of communicating processes. *Science of Computer Programming*, 5(2):171–199, 1984.
- [Bergstra et al., 1985] J.A. Bergstra, J.W. Klop, and J.V. Tucker. Process algebra with asynchronous communication mechanisms. In S.D. Brookes, A.W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency*, volume 197 of *Lecture Notes in Computer Science*, pages 76–95. Springer-Verlag, 1985.
- [Bergstra et al., 1994a] J.A. Bergstra, I. Bethke, and A. Ponse. Process algebra with combinators. In E. Börger, Y. Gurevich, and K. Meinke, editors, *Proceedings of CSL '93*, volume 832 of *Lecture Notes in Computer Science*, pages 36–65. Springer-Verlag, 1994.
- [Bergstra et al., 1994b] J.A. Bergstra, I. Bethke, and A. Ponse. Process algebra with iteration and nesting. *The Computer Journal*, 37(4):243–258, 1994.
- [Bergstra et al., 1994c] J.A. Bergstra, A. Ponse, and J.J. van Wamel. Process algebra with backtracking. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Proceedings of the REX Symposium "A Decade of Concurrency: Reflections and Perspectives"*, volume 803 of *Lecture Notes in Computer Science*, pages 46–91. Springer-Verlag, 1994.
- [Best, 1993] E. Best, editor. *Proceedings CONCUR 93*, Hildesheim, Germany, volume 715 of *Lecture Notes in Computer Science*. Springer-Verlag, August 1993.
- [Bezema and Groote, 1994] M.A. Bezema and J.F. Groote. Invariants in process algebra with data. In Jonsson and Parrow [1994], pages 401–416.
- [Blanco, 1995] J. Blanco. Definability with the state operator in process algebra. In Ponse et al. [1995], pages 218–241.
- [Bloom et al., 1988] B. Bloom, S. Istrail, and A.R. Meyer. Bisimulation can't be traced: Preliminary report. In *Conference Record of the 15<sup>th</sup> ACM Symposium on Principles of Programming Languages*, San Diego, California, pages 229–239, 1988. Full version available as Technical Report 90-1150, Department of Computer Science, Cornell University, Ithaca, New York, August 1990. Accepted to appear in *Journal of the ACM*.

- [Bol and Groote, 1991] R.N. Bol and J.F. Groote. The meaning of negative premises in transition system specifications (extended abstract). In J. Leach Albert, B. Monien, and M. Rodríguez, editors, *Proceedings 18<sup>th</sup> ICALP*, Madrid, volume 510 of *Lecture Notes in Computer Science*, pages 481–494. Springer-Verlag, 1991. Full version appeared as Report CS-R9054, CWI, Amsterdam, 1990.
- [Boudol *et al.*, 1990] G. Boudol, V. Roy, R. De Simone, and D. Vergamini. Process calculi, from theory to practice: verification tools. In Sifakis [1990], pages 1–10.
- [Brinksma, 1987] E. Brinksma. *Information processing systems – open systems interconnection – LOTOS – a formal description technique based on the temporal ordering of observational behaviour* ISO/TC97/SC21/N DIS8807, International Organization for Standardization, 1987.
- [Brookes *et al.*, 1984] S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560–599, 1984.
- [Caucal, 1990] D. Caucal. On the transition graphs of automata and grammars. Report 1318, INRIA, 1990.
- [Christensen, 1993] S. Christensen. *Decidability and decomposition in process algebra*. PhD thesis, University of Edinburgh, 1993.
- [Christensen *et al.*, 1992] S. Christensen, H. Hüttel, and C. Stirling. Bisimulation equivalence is decidable for all context-free processes. In Cleaveland [1992], pages 138–147.
- [Christensen *et al.*, 1993] S. Christensen, Y. Hirschfeld, and F. Moller. Bisimulation equivalence is decidable for basic parallel processes. In Best [1993], pages 143–157.
- [Cleaveland, 1992] W.R. Cleaveland, editor. *Proceedings CONCUR 92*, Stony Brook, NY, USA, volume 630 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.
- [Cleaveland *et al.*, 1990] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench. In Sifakis [1990], pages 24–37.
- [Copi *et al.*, 1958] I.M. Copi, C.C. Elgot, and J.B. Wright. Realization of events by logical nets. *Journal of the ACM*, 5:181–196, 1958.
- [De Simone, 1985] R. De Simone. Higher-level synchronising devices in MEIJE-SCCS. *Theoretical Computer Science*, 37:245–267, 1985.
- [Dershowitz, 1987] N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3(1):69–116, 1987.
- [Dershowitz and Jouannaud, 1990] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In *Formal Models and Semantics. Handbook of Theoretical Computer Science*, volume B, chapter 6, pages 243–320. Elsevier—MIT Press, Amsterdam, 1990.
- [Fokkink, 1994] W.J. Fokkink. The *tyft/tyxt* format reduces to tree rules.

- In M. Hagiya and J.C. Mitchell, editors, *Proceedings 2nd International Symposium on Theoretical Aspects of Computer Software (TACS'94)*, Sendai, Japan, volume 789 of *Lecture Notes in Computer Science*, pages 440–453. Springer Verlag, 1994.
- [Fokkink and Zantema, 1994] W.J. Fokkink and H. Zantema. Basic process algebra with iteration: completeness of its equational axioms. *The Computer Journal*, 37(4):259–267, 1994.
- [Giacalone et al., 1990] A. Giacalone, C.-C. Jou, and S.A. Smolka. Algebraic reasoning for probabilistic concurrent systems. In M. Broy and C.B. Jones, editors, *Proceedings IFIP Working Conference on Programming Concepts and Methods*, Sea of Galilee, Israel. North-Holland, 1990.
- [Glabbeek, 1987] R.J. van Glabbeek. Bounded nondeterminism and the approximation induction principle in process algebra. In F.J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, editors, *Proceedings STACS 87*, volume 247 of *Lecture Notes in Computer Science*, pages 336–347. Springer-Verlag, 1987.
- [Glabbeek, 1990] R.J. van Glabbeek. The linear time – branching time spectrum. In Baeten and Klop [1990], pages 278–297.
- [Glabbeek, 1993] R.J. van Glabbeek. The linear time – branching time spectrum II (extended abstract). In Best [1993], pages 66–81.
- [Glabbeek, 1995] R.J. van Glabbeek. On the expressiveness of ACP (extended abstract). In Ponse et al. [1995], pages 188–217.
- [Glabbeek and Weijland, 1989] R.J. van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics (extended abstract). In G.X. Ritter, editor, *Information Processing 89*, pages 613–618. North-Holland, 1989. Full version available as Report CS-R9120, CWI, Amsterdam, 1991.
- [Godskesen et al., 1989] J. Godskesen, K.G. Larsen, and M. Zeeberg. TAV–tools for automatic verification. Technical report R89–19, University of Aalborg, 1989.
- [Groote, 1990a] J.F. Groote. A new strategy for proving  $\omega$ -completeness with applications in process algebra. In Baeten and Klop [1990], pages 314–331.
- [Groote, 1990b] J.F. Groote. Transition system specifications with negative premises (extended abstract). In Baeten and Klop [1990], pages 332–341. Full version appeared as Technical Report CS-R8950, CWI, Amsterdam, 1989.
- [Groote and Ponse, 1994b] J.F. Groote and A. Ponse. The syntax and semantics of  $\mu$ CRL. In Ponse et al. [1995], pages 26–62.
- [Groote and Ponse, 1994] J.F. Groote and A. Ponse. Process algebra with guards: combining Hoare logic and process algebra. *Formal Aspects of Computing*, 6(2):115–164, 1994.



- [Groote and Vaandrager, 1992] J.F. Groote and F.W. Vaandrager. Structured operational semantics and bisimulation as a congruence. *Information and Computation*, 100(2):202–260, October 1992.
- [Hennessy, 1988] M. Hennessy. *Algebraic Theory of Processes*. MIT Press, Cambridge, Massachusetts, 1988.
- [Hoare, 1985] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, Englewood Cliffs, 1985.
- [Hoare et al., 1987] C.A.R. Hoare, I.J. Hayes, He Jifeng, C.C. Morgan, A.W. Roscoe, J.W. Sanders, I.H. Sorensen, J.M. Spivey, and B.A. Surfin. Laws of programming. *Communications of the ACM*, 30(8):672–686, 1987.
- [Jonsson and Parrow, 1994] B. Jonsson and J. Parrow, editors. *Proceedings CONCUR 94*, Uppsala, Sweden, volume 836 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.
- [Jouannaud and Kirchner, 1986] J.-P. Jouannaud and H. Kirchner. Completion of a set of rules modulo a set of equations. *SIAM Journal of Computing*, 15:1155–1194, 1986.
- [Jouannaud and Muñoz, 1984] J.-P. Jouannaud and M. Muñoz. Termination of a set of rules modulo a set of equations. In R. E. Shostak, editor, *7th International Conference on Automated Deduction*, volume 170 of *Lecture Notes in Computer Science*, pages 175–193. Springer-Verlag, 1984.
- [Kamin and Lévy, 1980] S. Kamin and J.-J. Lévy. Two generalizations of the recursive path ordering, 1980. Unpublished manuscript.
- [Kleene, 1956] S.C. Kleene. Representation of events in nerve nets and finite automata. In *Automata Studies*, pages 3–41. Princeton University Press, 1956.
- [Klop, 1992] J.W. Klop. Term rewriting systems. In *Handbook of Logic in Computer Science, Volume 2*, pages 1–116. Oxford University Press, 1992.
- [Klusener, 1993] A.S. Klusener. *Models and axioms for a fragment of real time process algebra*. PhD thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology, December 1993.
- [Koymans and Vrancken, 1985] C.P.J. Koymans and J.L.M. Vrancken. Extending process algebra with the empty process  $\epsilon$ . Logic Group Preprint Series Nr. 1, CIF, State University of Utrecht, 1985.
- [Larsen and Skou, 1992] K.G. Larsen and A. Skou. Compositional verification of probabilistic processes. In Cleaveland [1992], pages 456–471.
- [Lin, 1992] H. Lin. PAM: a process algebra manipulator. In K.G. Larsen and A. Skou, editors, *Proceedings of the 3rd International Workshop on Computer Aided Verification*, Aalborg, Denmark, July 1991, volume 575



- of *Lecture Notes in Computer Science*, pages 176–187. Springer-Verlag, 1992.
- [Mauw and Veltink, 1993] S. Mauw and G.J. Veltink, editors. *Algebraic Specification of Communication Protocols*. Cambridge Tracts in Theoretical Computer Science 36. Cambridge University Press, 1993.
- [Milne, 1983] G.J. Milne. CIRCAL: a calculus for circuit description. *Integration*, 1:121–160, 1983.
- [Milner, 1980] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.
- [Milner, 1983] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.
- [Milner, 1989] R. Milner. *Communication and Concurrency*. Prentice-Hall International, Englewood Cliffs, 1989.
- [Milner et al., 1992] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, Part I + II. *Information and Computation*, 100(1):1–77, 1992.
- [Moller and Tofts, 1990] F. Moller and C.M.N. Tofts. A temporal calculus of communicating systems. In Baeten and Klop [1990], pages 401–415.
- [Moller, 1989] F. Moller. *Axioms for concurrency*. PhD thesis, Report CST-59-89, Department of Computer Science, University of Edinburgh, 1989.
- [Nicollin and Sifakis, 1994] X. Nicollin and J. Sifakis. The algebra of timed processes, ATP: theory and application. *Information and Computation*, 114(1):131–178, 1994.
- [Park, 1981] D.M.R. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *5th GI Conference*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer-Verlag, 1981.
- [Peacock, 1830] G. Peacock. *A treatise of algebra*. Cambridge, 1830.
- [Plotkin, 1981] G.D. Plotkin. A structural approach to operational semantics. Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.
- [Ponse, 1991] A. Ponse. Process expressions and Hoare’s logic: showing an irreconcilability of context-free recursion with Scott’s induction rule. *Information and Computation*, 95(2):192–217, 1991.
- [Ponse, 1992] A. Ponse. Computable processes and bisimulation equivalence. Report CS-R9207, CWI, Amsterdam, January 1992. To appear in *Formal Aspects of Computing*.
- [Ponse, 1993] A. Ponse. Personal communication, June 1993.
- [Ponse et al., 1995] A. Ponse, C. Verhoef, and S.F.M. van Vlijmen, editors. *Algebra of Communicating Processes, Utrecht 1994*, Workshops in Computing. Springer-Verlag, 1995.

- [Ritchie and Thompson, 1974] D.M. Ritchie and K. Thompson. The UNIX time-sharing system. *Communications of the ACM*, 17(7):365–375, 1974.
- [Sifakis, 1990] J. Sifakis, editor. *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France, June 1989, volume 407 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.
- [Tofts, 1990] C.M.N. Tofts. A synchronous calculus of relative frequency. In Baeten and Klop [1990], pages 467–480.
- [Troeger, 1993] D.R. Troeger. Step bisimulation is pomset equivalence on a parallel language without explicit internal choice. *Mathematical Structures in Computer Science*, 3:25–62, 1993.
- [Vaandrager, 1990a] F.W. Vaandrager. Process algebra semantics of POOL. In Baeten [1990], pages 173–236.
- [Vaandrager, 1990b] F.W. Vaandrager. Two simple protocols. In Baeten [1990], pages 23–44.
- [Vaandrager, 1993] F.W. Vaandrager. Expressiveness results for process algebras. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Proceedings REX Workshop on Semantics: Foundations and Applications*, Beekbergen, The Netherlands, June 1992, volume 666 of *Lecture Notes in Computer Science*, pages 609–638. Springer-Verlag, 1993.
- [Veltink, 1993] G.J. Veltink. The PSF toolkit. *Computer Networks and ISDN Systems*, 25:875–898, 1993.
- [Verhoef, 1992] C. Verhoef. *Linear Unary Operators in Process Algebra*. PhD thesis, University of Amsterdam, June 1992.
- [Verhoef, 1994a] C. Verhoef. A congruence theorem for structured operational semantics with predicates and negative premises. In Jonsson and Parrow [1994], pages 433–448.
- [Verhoef, 1994b] C. Verhoef. A general conservative extension theorem in process algebra. In E.-R. Olderog, editor, *Programming Concepts, Methods and Calculi (PROCOMET '94)*, volume A-56 of *IFIP Transactions A: Computer Science and Technology*, pages 149–168. Elsevier, 1994.
- [Vrancken, 1986] J.L.M. Vrancken. The algebra of communicating processes with empty process. Report FVI 86-01, Dept. of Computer Science, University of Amsterdam, 1986.
- [Weijland, 1989] W.P. Weijland. *Synchrony and asynchrony in process algebra*. PhD thesis, University of Amsterdam, 1989.

# Correspondence between operational and denotational semantics: the full abstraction problem for PCF\*

C.-H. L. Ong

---

## Contents

1	Introduction . . . . .	270
1.1	Relating operational and denotational semantics . . . . .	270
1.2	Full abstraction problem for PCF . . . . .	274
1.3	Quest for a solution: a survey . . . . .	276
1.4	Organization of the chapter . . . . .	280
1.5	A selected bibliography . . . . .	281
2	A Programming language for computable functions . . . . .	282
2.1	The programming language PCF . . . . .	283
2.2	Syntax of the reduction system $\lambda_{\vec{\gamma}}^{\rightarrow}(\mathbf{A})$ . . . . .	284
2.3	Reduction rules of the system $\lambda_{\vec{\gamma}}^{\rightarrow}(\mathbf{A})$ . . . . .	287
2.4	Operational properties of $\lambda_{\vec{\gamma}}^{\rightarrow}(\mathbf{A})$ . . . . .	288
2.5	Böhm trees and the Syntactic Continuity Theorem . . . . .	292
2.6	Sequentiality and stability . . . . .	295
2.7	The programming language PCF . . . . .	299
3	Operational and denotational semantics of PCF . . . . .	301
3.1	Context Lemma and observational extensionality . . . . .	302
3.2	Denotational models . . . . .	304
3.3	Adequacy . . . . .	308
3.4	Order-extensional, continuous, fully abstract model . . . . .	313
3.5	Full abstraction and non-full abstraction results . . . . .	319

---

\*Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD; and Department of Information Systems and Computer Science, National University of Singapore, Lower Kent Ridge Road, Singapore 0511. This work was done when the author was a research fellow at the Computer Laboratory, University of Cambridge.

4	Towards a characterization of PCF-sequentiality . . . . .	325
4.1	Concrete data structures and sequential functions . . . . .	327
4.2	dI-domains and stable functions . . . . .	335
4.3	Sequential algorithms . . . . .	339
4.4	Observable algorithms and PCF with error values . . . . .	342
4.5	Towards a characterization of sequentiality . . . . .	345

This chapter concerns the relationship between operational semantics and denotational semantics of higher-type sequential functional programming languages. We choose Scott's language PCF as the vehicle for developing the leading ideas and focus on a line of research inspired by the full abstraction problem for PCF.

## 1 Introduction

This chapter addresses a number of computationally relevant mathematical problems arising from the relationship between operational and denotational semantics of higher-type (or higher-order) sequential functional programming languages. We choose Scott's language PCF (which stands for Programs for Computable Functions) as the vehicle for developing the leading ideas.

The operational semantics of a programming language is always given, either explicitly or implicitly, with reference to a specific machine which is usually abstract and idealized. According to operational semantics, the meaning of a program is expressed in terms of the observable behaviour of the machine when it is running the program. A formalism known as Natural Semantics [Kahn, 1988] (for a good example, see [Milner *et al.*, 1990]), which is based on Plotkin's Structural Operational Semantics [Plotkin, 1981b], is very widely used to express operational semantics of programming languages. Denotational semantics is a way of giving meanings to programs in the style of Tarski. A mathematical structure is first chosen as the universe of discourse. This structure is referred to as the denotational model. A semantic or valuation function then maps programs to denotations which are elements of the model. An important characteristic of the denotational approach is the definition of the valuation function by structural induction. This gives rise to a *compositional* semantics of programs: the meaning of a program is defined in terms of the respective meaning of its components.

### 1.1 Relating operational and denotational semantics

Each semantic style induces an (equational) theory of program equivalence. According to operational semantics, two programs are equivalent if the machine running the two programs is observed to behave in the same way.



In contrast, according to denotational semantics (see, for example, Mosses' survey article [Mosses, 1990]), two programs are equivalent whenever they denote the same object in the denotational model. Since the two styles of prescribing semantics are so conceptually dissimilar, there is no *a priori* reason why the respective theories of program equivalence should coincide. Therefore, it is a meaningful question to ask how one theory of program equivalence relates to the other. This is so at least from the conceptual and philosophical point of view: in fact, the further apart the two semantics are in terms of conception and style, the more pointed it is as a question and the more reason there is to seek a correspondence. (For if the two styles were motivated by the same set of considerations, it would be absurd to find two different induced theories of program equivalence.) There is another more pragmatic and technological reason why it is important to study the correspondence between operational and denotational semantics. It has to do with the aims and uses of both semantics. The primary aim of denotational semantics is to enable a canonical definition of the meanings of programs. A canonical definition is important for it not only documents the design of the language, but also establishes a mathematically precise standard for implementation of the language on computers. Without a secure foundation for the meanings of programs, there can be no basis for reasoning about the correctness and other properties of programs. In contrast, operational semantics is well-suited to capturing the intuitions of a programming language at a level much closer to the machine than the programmer. It is ideal for describing low-level implementations rather than high-level specifications of a programming language. In summary, each semantic style is good for something. Therefore, we want to make use of both styles so that one may complement the other. This brings us to a crucial point: in order to benefit from *both* semantic styles, it is necessary to know how one semantics relates to the other.

Let us examine each of the two styles of program equivalence more closely. What does it mean to say that two programs (or program fragments) have the same observable behaviour when executed on the same machine? Programmers understand this notion of sameness well: two program fragments are equivalent if they can always be *interchanged* without affecting the visible or observable outcome of the computation. This criterion of sameness which is called *observational equivalence* is formally expressed in terms of invariance of observable outcome under all program contexts. This definition hinges on two things about the programming language in question: first, the notion of observable outcome or the observables, and secondly, program contexts. The *observables* are computational phenomena occurring at the conclusion of a computation which can be unambiguously observed by the programmer. Examples: a computation halts with the numeral 3 being displayed on the computer screen; the printer "out of paper" warning light comes on, *etc.* Program contexts are an auxiliary syntac-



tic construction; they are not objects that computers manipulate, but a conceptual device. Intuitively, they are programs with typed “holes” into which type-compatible program fragments may fit.

**Overview of PCF** In order to give a precise definition of observational equivalence, we need some information about the programming language in question. PCF was first introduced by Scott in a famous paper [Scott, 1969] which remained an unpublished manuscript until its recent appearance in the Böhm Festschrift [Scott, 1993]. Scott intended PCF to be a “logical calculus (or algebra)” for the purpose of studying computability and logical properties of programs by using type theory. Plotkin presented PCF explicitly as a programming language in a seminal paper entitled *PCF as a Programming Language* [Plotkin, 1977]. The papers of Scott and Plotkin could be said to be the main inspiration behind the body of work surveyed in this chapter. Two other independent but apparently lesser known sources should be mentioned: namely, the work of Sazonov [Sazonov, 1975; Sazonov, 1976] and the work of Kleene (see e.g. [Kleene, 1978]). PCF is just the simply typed lambda calculus augmented by general recursion in the form of a fixed-point operator at every type, and further augmented by basic arithmetic. Types of PCF are built up from the two ground types ( $\iota$  for natural numbers and  $o$  for booleans) by using a binary type constructor  $\Rightarrow$ , usually called the arrow or function type. Programs of the language are closed terms of ground type, and (ground) values are the numerals and the booleans. Following the function paradigm, to compute a program is to evaluate it, i.e. to ascertain its value, and evaluation is implemented by a process of reduction. We shall write  $M \Downarrow V$  to mean that “the program  $M$  evaluates (or reduces) to the value  $V$ ”. Writing program contexts as  $C[X]$  where the “hole” represented by  $X$  is to be thought of as a kind of meta-variable, we can now express observational equivalence in more precise terms.

Two program fragments  $M$  and  $N$  are said to be *observationally equivalent*, which we write as  $M \approx N$ , if for every type-compatible program context  $C[X]$  such that both  $C[M]$  and  $C[N]$  are programs, and for any value  $V$ ,  $C[M] \Downarrow V$  if and only if  $C[N] \Downarrow V$ .

As usual,  $C[M]$  means “the program which is obtained from the context  $C[X]$  by substituting  $M$  for every occurrence of  $X$  in  $C[X]$ ”.

In [Scott, 1969], Scott introduced what has come to be known as the Scott-continuous function space model for PCF-programs. He used the standard flat CPOs to interpret booleans and the natural numbers respectively, and Scott-continuous function space to interpret function types. We shall call this model the *standard continuous model* (or simply, the *standard model*) of PCF. The adjective “continuous” qualifies the way fixed-point operators are interpreted, i.e. standardly as the least upper bound of the

$\omega$ -chain of successive iterates, see e.g. [Tarski, 1955]. Note that the standard model is *order-extensional*<sup>1</sup> in the sense that elements of domains which interpret function types are functions and they are ordered point-wise. More generally, writing the denotation of a program  $M$  as  $\mathcal{A}[M]$ , we say that the denotational semantics  $\mathcal{A}[-]$  is *adequate* if for every pair of type-compatible terms  $M$  and  $N$ ,

$$\mathcal{A}[M] = \mathcal{A}[N] \quad \implies \quad M \approx N.$$

If, in addition, the converse is also valid, that is to say,

$$\mathcal{A}[M] = \mathcal{A}[N] \quad \iff \quad M \approx N,$$

then the denotational semantics is said to be (equationally) *fully abstract* for the language. The notion of full abstraction is due to Milner [Milner, 1975], though it seems implicit in work in the pure lambda calculus by Hyland, Morris [Morris, 1968], Plotkin, Wadsworth [Wadsworth, 1976; Wadsworth, 1978] and others. Adequacy and full abstraction tell us how well the operational and the denotational views of program equivalence relate to each other. They are indications of how reliable or how “fitting” the denotational model is in relation to the language. More specifically, adequacy assures us that the model is reliable enough for affirming observational equivalence between two terms since denotational equality suffices; but the model is not necessarily reliable for refuting equivalence for which we need full abstraction. Adequacy is usually relatively easy to establish, but this is not so for full abstraction. A model is not fully abstract usually because it is in some sense too rich a structure for the language: it contains semantic objects which “cannot be computed” by the programming language. Conversely, a model which is fully abstract for a language provides a very satisfactory characterization of (the observational equivalence of) the language in terms of the denotational model.

Plotkin showed in [Plotkin, 1977] that the standard model is adequate but not fully abstract for PCF. He also pointed out the reason for the failure of full abstraction. The mismatch may be explained, in a nutshell, by the fact that while PCF-programs correspond to “sequential” algorithms (Section 2 makes this precise), the standard Scott-continuous function space model contains “parallel” functions, or more accurately, functions which can only be implemented by parallel algorithms. This point was made explicit by the main result of *op. cit.* as follows.

**Theorem 1.1.1 (Plotkin).** *The standard Scott-continuous function space*

---

<sup>1</sup>More precisely, this means that for any partially ordered domains  $D^\sigma$  and  $D^\tau$  which interpret the types  $\sigma$  and  $\tau$  respectively, and for any elements  $f, g$  of  $D^{\sigma \Rightarrow \tau}$ ,  $f \leq g$  if and only if  $f(a) \leq g(a)$  for every  $a \in D^\sigma$ .

*model is fully abstract for a programming language which is obtained by extending PCF with a parallel conditional constant.* ■

It is known that any continuous, order-extensional model of PCF which follows the standard<sup>2</sup> interpretation is necessarily a system of Scott domains, *i.e.*  $\omega$ -algebraic, consistently complete CPOs (see Theorem 3.4.3). Further, there is a necessary and sufficient condition for any such model to be fully abstract for PCF. The condition is that all compact elements of the model be PCF-definable (see Theorem 3.4.2). Plotkin's result leaves open the question of whether there is a denotational model which is fully abstract for PCF proper. This was quickly answered by Milner [Milner, 1977]:

**Theorem 1.1.2 (Milner).** *There is a unique (up to isomorphism) order-extensional, continuous, inequationally fully abstract model for PCF.* ■

## 1.2 Full abstraction problem for PCF

What then is the full abstraction problem for PCF? While there is widespread agreement that the problem is a difficult one, there is much less consensus as to what constitutes a solution to the problem. At one level, this question seems superfluous; for we already know that there is a unique fully abstract model for PCF — witness Milner's construction. This question has to do with the philosophical question of what a good model is. A good model enlightens; it gives a new perspective of the behaviour or operational semantics of the programming language in question. There is no doubt that Milner's result settles an important question and his construction is a valuable contribution. Nonetheless, because his construction is essentially a term model, it is arguable that the model does not much increase our understanding of PCF beyond what can already be gleaned directly from the syntax of the language. One way to formulate the problem which, we believe, strikes at the root of the issue is the following:

**The full abstraction problem for PCF** *Give an abstract, synthetic account of the unique order-extensional, continuous, inequationally fully abstract model of PCF as identified by Milner.* ■

It is worth expanding on the two operative words. By an *abstract* model, we mean a model which is constructed without recourse to the syntax or operational semantics of the language. In fact, the more computationally neutral the model is in its conception, the more apposite it is as a solution. By *synthetic* description is meant a constructive, axiomatic explanation of the function space which interprets the PCF-types (in terms of the respective interpretation of the components). To illustrate the twin criteria of abstract and synthetic description, consider first the standard

---

<sup>2</sup>An interpretation of PCF is said to be *standard* if the ground types are interpreted as the respective flat CPOs.



Scott-continuous function space model of PCF. This model evidently satisfies the criterion of abstract construction: CPOs and continuous functions are computationally neutral semantic objects. The synthetic description of the model is also satisfactory in every way: Scott domains and continuous functions form a cartesian closed category. Continuous functions are of course by definition order-extensional, but unfortunately, some parallel algorithms (like “parallel- or”) which are not PCF-definable have denotations as continuous functions. So the continuous function space model fails to characterize higher-type sequentiality in the PCF sense because it is too “large”. Next, consider the construction of Mulmuley [Mulmuley, 1986] which achieved a fully abstract characterization of PCF by a syntax-based quotienting operation that cuts the somewhat non-standard (base types are interpreted as flat lattices as opposed to flat CPOs) continuous function space model down to size. Such an approach, in our view, falls short of the criterion of synthetic construction.

Since the crux of the full abstraction problem is the characterization of *sequential* computations, we may reformulate the full abstraction problem for PCF as the problem of finding an abstract, synthetic characterization of higher-type, sequential, PCF-definable function(al)s. Formulated in this way, we highlight the epistemological difficulties inherent to the problem, for we do not have a proper definition of higher-type sequentiality from first principles. At any rate, to date there is certainly no notion of higher-type sequentiality which can be said to be canonical in any sense. In fact, it is unclear whether there are various inequivalent notions of higher-type sequentiality, all of which equally appealing; or like the notion of effective computability, there is essentially one idea under possibly different guises.

**Some criteria** The full abstraction problem for PCF in the above qualitative sense is by its nature incapable of being precisely specified because the underlying considerations are philosophical rather than mathematical, and so more or less subjective in nature. Therefore, it seems all the more important to lay down a few criteria which should be as objective as possible so that progress in the solution to the problem may be calibrated to some extent and be seen in perspective.

A continuous model of PCF is a CPO-enriched cartesian closed category whose canonical operations such as composition, pairing and currying are continuous and satisfy some strictness conditions appropriate to the operational semantics of PCF. In view of Theorem 3.4.2, we may say that the full abstraction problem for PCF boils down to the following:

**Intensional full abstraction for PCF** *Find a CPO-enriched cartesian closed category of Scott domains (satisfying continuity and strictness constraints appropriate to the operational semantics) such that all of whose compact elements are PCF-definable.*



The qualification of intensionality (which is due to Abramsky) is apropos since there is no intrinsic reason why the denotation of a PCF-program in such a fully abstract model should be a function.

In [Jung and Stoughton, 1993], Jung and Stoughton seek “a weak but precise minimal condition that a semantic solution of the full abstraction should satisfy”. The second criterion, which we call the *Jung-Stoughton criterion*, imposes an effectivity constraint on the way the fully abstract model is presented. It requires an effective construction of the fully abstract model restricted to *finitary* PCF, *i.e.* that part of PCF which is generated solely from constants of boolean type.

The third criterion is the hardest both to state precisely and to satisfy. It asks for an axiomatic characterization of higher-type, PCF-sequential functions. By way of comparison, if it is right to think of the first two criteria as contributions to the representation theory of higher-type sequentiality, then the third is in the business of giving it a *definition*. One appealing way to characterize PCF-definable functions is to express them in terms of some preservation property in an order-theoretic framework, for example in the style of Bucciarelli and Ehrhards’ strongly stable functions [Bucciarelli and Ehrhard, 1991]. Another way is to characterize them topologically say, as a refinement of the Scott topology (in the sense of *e.g.* [Brookes and Geva, 1992b]). Such an approach is likely to be very hard, if it is at all feasible.

### 1.3 Quest for a solution: a survey

This sets the scene for a line of research motivated by the quest for a solution to the full abstraction problem (in the qualitative sense) for PCF. As Plotkin already intimated in [Plotkin, 1977], the key to the solution is an abstract characterization of sequential computation. To do that, one needs a proper understanding of sequentiality. The matter is straightforward in the case of first-order computation. Milner [Milner, 1977] and others have already obtained a number of satisfactory abstract descriptions of first-order sequential functions. Intuitively, the meaning of sequential computation is clear enough: it is to do “one thing at a time” at any intermediate stage of the computation, and possibly in a specific order. The real difficulty lies in describing sequential, functional computation at higher type.

The first major contribution was made by Kahn and Plotkin and reported in a technical report written in French [Kahn and Plotkin, 1978]. Like the papers of Scott and Plotkin mentioned above, a revised version of this paper [Kahn and Plotkin, 1993] in English has also appeared in the recently published Böhm Festschrift. They introduce a class of mathematical structures known as *concrete data structure* (CDS). CDS is an elaborate structure specially designed to articulate sequential computations. The framework of CDS and the Kahn-Plotkin sequential function is a highly innovative conceptual advance in understanding higher-type sequentiality.



However, their framework does not give rise to a cartesian closed category. This is not surprising since Kahn and Plotkin did not aim to carry out a systematic analysis of higher-type functional computation in that paper. Rather, their primary objective was to examine the behaviour of stream-like computations.

The search for a cartesian closed category of “sequential functions” became the focus of research. Historically, the research bifurcated at this juncture. The sticking point lies in an apparent tradeoff between the two essential features: on the one hand, sequentiality, which refers to a computational process extended over time, is an inherently intensional notion; and on the other, the requirement that such computational processes interact with each other in a functional, or extensional, way. So to characterize sequential functions is to find an appropriate setting in which both properties can be held in tension. Unfortunately, based on the work of Berry and Curien in the late 1970’s, it would seem that in order to get a cartesian closed category of “sequential functions”, one of the two criteria has to give.

One major effort consisted in relaxing the constraints of sequentiality but staying within the framework of functions. This led Berry to the notion of *stability* [Berry, 1978b; Berry, 1979]. The appropriate morphisms are stable functions which are continuous functions that preserve greatest lower bounds of consistent (or “upper bounded”) subsets; and the objects are dI-domains—Scott domains which satisfy a distributivity property and axiom (I) (which says that every compact element dominates finitely many elements). Stable functions are not ordered by the standard extensional (or pointwise) ordering<sup>3</sup> but by another ordering called stable ordering. A major result is that the category of dI-domains and stable functions is cartesian closed.

The other approach builds on the central ideas behind the framework of CDS and the Kahn-Plotkin sequential function but sacrifices extensionality. Thus, Berry and Curien introduced *sequential algorithms* over CDSS [Berry and Curien, 1982] (see also Curien’s book [Curien, 1993a] for a comprehensive introduction). Sequential algorithms may be thought of as intensional refinements or “implementations” of Kahn-Plotkin sequential functions. There are two reasons why this way of thinking is appropriate. First, it is possible to express each sequential algorithm as a pair of the form  $(f, \phi)$  where  $f$  is just a Kahn-Plotkin sequential function, and  $\phi$ , referred to as the associated computation strategy, is a partial function

---

<sup>3</sup>Care should be taken not to confuse the two concepts: *extensional* object and *order-extensional* functions. We use the adjective *extensional* simply to mean the property of being a function as opposed to, say, an algorithm which is an intensional thing. However, even if the morphisms of an order-enriched category are extensional, they need not necessarily be *order-extensional*, i.e. ordered in the pointwise fashion. The category of dI-domains and stable functions is a case in point.

that chooses a sequentiality index at each stage of the computation. Secondly, it is a theorem that the quotient of the CPO of sequential algorithms by the extensional equality is isomorphic to the CPO of Kahn-Plotkin sequential functions with respect to the stable ordering. Remarkably, unlike Kahn-Plotkin sequential functions, sequential algorithms do give rise to a cartesian closed category.

Each of the approaches gives rise to a CPO-enriched cartesian closed category and provides a continuous model for PCF but neither leads to a solution of the full abstraction problem for PCF. In the case of the stable function space model, a simple reason is that the ordering in question is not the extensional ordering but rather the stable ordering. In the case of the model associated with sequential algorithms, the morphisms are not even functions<sup>4</sup>.

Recently, drawing on their intuitions as programmers, Cartwright and Felleisen [Cartwright and Felleisen, 1992; Cartwright *et al.*, 1994] introduced a continuous, order-extensional model for PCF which is based on what they call *observably sequential functions*. Curien [Curien, 1992] immediately realized that the observably sequential functions were a natural *extensional* refinement of sequential algorithms. This is remarkable because the sequential algorithms being considered in the extended setting, which are called *observable algorithms*, are still very much intensional in nature, and are most succinctly represented as a kind of decision trees. The key to this surprising development is that the concrete data structures are now equipped with “error values”. To ensure a well-behaved mechanism of function application, observable algorithms are required to “percolate errors to the top” when they are applied to arguments. A main result is that the category of deterministic CDSs with error values and observable algorithms is cartesian closed. The associated model is not fully abstract for PCF, but it is for a language called SPCF which is PCF extended by error values and escape handling control facilities which resemble the *catch* facility in some versions of the programming language Lisp. Curien *et al.* also showed that the original model of sequential algorithms is fully abstract for PCF extended with *catch* only. Though the concept of error values is instrumental to the proof of full abstraction, sequential algorithms are definable without using errors.

Kahn-Plotkin sequentiality and Berry and Curiens’ sequential algorithm are both formulated within the rather concrete setting of CDS. Kahn, Plotkin, Winskel [Winskel, 1980; Winskel, 1987] and others have proved

---

<sup>4</sup>A “deeper” explanation has to do with a subtle point about the extensional way in which PCF functionals interact with function arguments. Curien has shown that there is no PCF-term of the type  $(o \Rightarrow o \Rightarrow o) \Rightarrow o$  which distinguishes between left-strict or and right-strict or (say). In contrast, sequential algorithms are capable of discriminating between two computations which implement the same function but differ *intensionally* as in the above sense.

various representation theorems for CDSS. One result shows that this approach applies to a (somewhat) restricted class of CPOs known as *concrete domains*. Can these two leading ideas in the understanding of higher-order sequentiality be generalized to a more abstract setting? In a series of papers [Bucciarelli and Ehrhard, 1991; Bucciarelli and Ehrhard, 1993b; Ehrhard, 1994b], Bucciarelli and Ehrhard set out to answer these questions systematically. They propose an abstract framework called *sequential structure* which is a pair  $\langle X_*, X^* \rangle$  where

- $X_*$ , the collection of “data” or “answers”, is a dI-domain, and
- $X^*$ , the collection of “questions”, is a kind of linear map from  $X_*$  to the two-point dI-domain  $(\perp < \top)$ . An element of  $X^*$  should be thought of as a linear property of elements of  $X_*$ .

Think of a sequential structure as a concrete data structure made abstract. Their key idea was to replace cells by a class of linear maps. States of a CDS then correspond to points of the data space  $X_*$ . Remarkably, in this abstract setting, sequential algorithms can be defined quite naturally as pairs of the form  $(f, \phi)$  where  $f$ , a sequential function, describes the input-output behaviour of the algorithm; and  $\phi$ , a partial function, describes its intensional properties. The enabling relation in a CDS which formalizes a notion of “immediate reachability” or “adjacency in the ordering” also has a natural, abstract representation in the setting of sequential structure. Ehrhard and Bucciarelli show that a cartesian closed category of sequential structures with enabling and sequential algorithms can be constructed; and furthermore, into this category, the category of deterministic CDSS (DCDSS) and sequential algorithms can be fully and faithfully embedded. Thus, the goal of extending sequential algorithms to an abstract setting is achieved.

Bucciarelli and Ehrhard [Bucciarelli and Ehrhard, 1991; Bucciarelli, 1993] also introduced the notion of *strong stability*. They were motivated by the observation that for DCDSS, Kahn-Plotkin sequential functions can be given an equivalent description in more algebraic terms. According to this definition, a sequential function is a continuous function preserving a certain class of meets. They then cast this idea in an abstract setting. The “domains” are dI-domains  $D$  equipped with a collection  $\mathbf{C}(D)$  of finite subsets of  $D$  satisfying a number of axioms. Call the collection  $\mathbf{C}(D)$  a *coherence* and any of its elements a *coherence property*. A continuous function  $f : D \rightarrow E$  between dI-domains with coherence is said to be *strongly stable* if

- it preserves coherence properties, i.e.  $f(A)$  is in  $\mathbf{C}(E)$  whenever  $A$  is in  $\mathbf{C}(D)$ , and
- it preserves greatest lower bounds of coherence properties, i.e.  $f(\sqcap A) = \sqcap f(A)$  for any  $A$  in  $\mathbf{C}(D)$ .

Their result is that the category of dI-domains with coherence and strongly



stable functions is cartesian closed. It is known that the associated model is not fully abstract for PCF; but how close does it model “sequential functions”? At first order, strong stability coincides with Kahn-Plotkin sequentiality. However, at higher order, we find ourselves at a loss conceptually for we are faced with the same fundamental question which we came across earlier: is there a *standard* or canonical definition for higher-order sequentiality?

Persisting in the background of these developments is a deeper, more philosophical question of whether there is such a thing as a *canonical* notion of sequential computation at higher type. Clearly, the kind of computation *defined* by PCF is at least a contender for such a standard. But it seems to us that there is no compelling evidence (yet) that PCF-style computation is the only acceptable notion of higher-order sequentiality. This is closely connected with a problem which Kleene posed in [Kleene, 1978].

**Kleene’s problem.** Find “a class of functions which shall coincide with all the partial functions which are ‘computable’ or ‘effectively decidable’, so that Church’s 1936 Thesis will apply with the higher types included”.

In fact in this paper, Kleene initiated what is in effect an attack on the full abstraction problem for PCF. The series of four papers by Kleene [Kleene, 1978; Kleene, 1980; Kleene, 1982; Kleene, 1985] are all concerned with an attempt to give meaning to PCF (or rather to Kleene’s own preferred version of Platek’s recursion in terms of schemes) in terms of rules for a kind of dialogue game. Kleene’s idea of a dialogue developed in parallel with and independently of the work on CDSS. Kleene’s initiative was followed up by Robin Gandy and his student Giovanni Pani. Their work [Gandy, 1993] is not published, but they have investigated a number of possible approaches and have accumulated numerous (counter-)examples.

These questions of higher-order sequentiality and Kleene’s seemingly more general question are of central importance to Computer Science. They deserve further investigations.

## 1.4 Organization of the chapter

We have written this chapter in the style of a survey article. Our aim is to give a bird’s eye view of the body of knowledge inspired by the full abstraction problem for PCF: we therefore emphasize conceptual developments at the expense of technical details. Most of the results mentioned in the chapter are not proved. This, we hope, is to some extent compensated for by the provision of detailed bibliographical references.

In Section 2, we begin by giving a historical introduction of the programming language PCF. The general aim is to study syntactic and operational properties of PCF. The high points of the section are the properties of stability and sequentiality which are shown to be satisfied by PCF. In order to

state the full abstraction problem precisely, it is necessary to clarify what constitutes a denotational model of PCF. This is dealt with at the start of Section 3. The Scott-continuous function space model of PCF is then presented. The main theme of this section is the foundational work on PCF by Plotkin, Milner and others. This leads up to a more or less precise statement of the full abstraction problem for PCF. Section 4 surveys the various attempts at solving the full abstraction problem. The main topics are: Kahn and Plotkin's concrete data structure and the associated notion of sequential function, Berry and Curien's sequential algorithms, Berry's stable functions and dI-domains, Cartwright, Felleisen and Curien's work on observable algorithms and the extension of PCF with error handling, Bucciarelli and Ehrhard's recent work on sequential structure and strong stability.

## 1.5 A selected bibliography

To the best of our knowledge, there are only two survey papers (excluding this one) covering grounds similar to this chapter, namely [Berry *et al.*, 1986] and [Curien, 1992]. The first gives an exposition of the "state of the art" up to Berry and Curien's work on sequential algorithms. The second, based on an invited talk given at the 1991 Durham Symposium, complements the first and brings it more or less up to date. A short paper of Meyer and Cosmadakis [Meyer and Cosmadakis, 1988] gives a highly readable introduction to full abstraction problems in general. An excellent book-length treatment of most of the material surveyed in this chapter is the second edition of Curien's book [Curien, 1993a]. Stoughton's book [Stoughton, 1988] presents a language-independent theory of fully abstract semantics of programming languages.

Barendregt's treatise [Barendregt, 1984] on the  $\lambda$ -calculus has an encyclopedic coverage of the syntactic properties of the calculus including stability and sequentiality. Proofs of most of the results in Section 2 can be found there. For the modicum of category theory used in the chapter, MacLane's book [MacLane, 1971] more than suffices, using perhaps [Kelly, 1982] as a (very) occasional supplement. Gunter's [Gunter, 1992] and Winskel's [Winskel, 1993] textbooks on the semantics of programming languages cover part of the material in Section 3 (*e.g.* the standard results on PCF) very carefully. They also provide a good introduction to denotational semantics. The best reference for Plotkin's pioneering contribution is none other than his own paper [Plotkin, 1977]. Finally, we recommend Bucciarelli's recent PhD thesis [Bucciarelli, 1993] for a comprehensive and detailed account of most of the material covered in Section 4. This includes, in particular, a very readable account of his recent joint work with Ehrhard and a line of research pursued by Kleene (spanning some seven years) which deserves to be better known by computer scientists interested



in the semantics of programming languages. More detailed bibliographical remarks (especially those pointing out where detailed proofs of results may be found) are provided throughout the chapter.

## 2 A Programming language for computable functions

**Historical remarks**<sup>5</sup> In the early 1940's, Gödel considered a notion of primitive recursive functionals of finite type, which we now call Gödel's System **T**, in connection with what came to be known as the Dialectica Interpretation [Gödel, 1958; Gödel, 1990]. Gödel presented his results as a contribution to a *liberalized* version of Hilbert's programme.<sup>6</sup> Gödel's work was later extended to the bar recursive functionals by Spector [Spector, 1962] who used them to give a constructive consistency proof for classical analysis. However, the first full-blown generalization of ordinary recursion theory to higher types was made by Kleene in the late 1950's (see [Kleene, 1959; Kleene, 1963] for a formulation in terms of computation schemes). In Kleene's theory, a notion of partial recursive function of higher type is defined over a total type structure. Recursion is introduced by a computation scheme which essentially encapsulates the Second Recursion Theorem. In this theory, partial recursive functions are not closed under substitution, and a natural formulation of the First Recursion Theorem fails. Kleene exhibited these features in [Kleene, 1963] where he also observed in passing that a theory involving the application of (partial) functions to partial functions might be possible.

An attempt was made by Platek in his thesis [Platek, 1966] to develop a recursion theory on partial functions of higher type which avoids the problems of the Kleene theory. The type structure Platek considers is that of hereditarily order-preserving functions over what we now call the flat CPO of natural numbers. Platek couched his thesis in terms of computation schemes, and recursion is introduced by a scheme amounting to the First Recursion Theorem. The essentials of the theory are definition by cases and least fixed points, and Platek does introduce a  $\lambda$ -calculus formulation which can be regarded as a precursor to PCF.

---

<sup>5</sup>The discussion on the historical background of PCF is based extensively on unpublished notes of Martin Hyland.

<sup>6</sup>Hilbert had set out to justify classical mathematics systematically in terms of notions which should be as intuitively clear as possible. A focus of his programme was the consistency of classical number theory; he wanted to find these intuitively clear notions in the domain of "finitary mathematics". Bernays subsequently pointed out that in order to prove the consistency of classical number theory, it was necessary to extend Hilbert's finitary standpoint by admitting "abstract concepts" of a certain kind in addition to the combinatorial concepts relating to symbols. Gödel introduced System **T**, which is essentially the simply typed  $\lambda$ -calculus augmented by primitive recursion, as a vehicle for expressing the "abstract notion" which is the key to making Hilbert's programme (in the modified sense) viable.

The formal system PCF was introduced by Scott in a famous paper [Scott, 1969] which remained an unpublished manuscript for a long time until its recent appearance in the Böhm Festschrift [Scott, 1993]. The syntax<sup>7</sup> of PCF is simple enough: it is essentially the simply typed  $\lambda$ -calculus augmented by general recursion in the form of a fixed-point operator at every type, and further augmented by basic arithmetic. Scott intended PCF to be a “logical calculus (or algebra)” for studying computability and other logical properties of programs by using type theory [Church, 1940].

A major theme of Scott’s work is the relationship between the *logical* types which are the higher types, and the data types which are the ground and first-order types. The former is used to study the latter; the theory of data types requires the higher-type objects—the computable functionals—for its formalization, as emphasized by Scott. Of course, Scott had in mind a semantics in terms of what we now call Scott-continuous functions and for this interpretation, a quite straightforward operational semantics is appropriate: we can think of all the computations as being finite. Hence, the theory introduced by Scott is in principle implementable and has been the focus of much attention in Computer Science. Another theme in Scott’s paper is completeness about which he asked several questions. One such question concerns the power of expression of the language with respect to the continuous function space model; in a word, definability. What came to be known as the full abstraction problem for PCF was thus adumbrated: Scott observed that parallel or is not definable in the language and that an implementation on a “sequential machine” would require a dovetailing strategy. System **T** and PCF (as a formal system in the sense of Scott’s original presentation as opposed to a programming language) are similar in various ways, though there is an important difference: Scott’s approach admits partial functions whereas Gödel’s is only concerned with total functions.

## 2.1 The programming language PCF

In [Plotkin, 1977], Plotkin presented PCF explicitly as a programming language and studied the relationship between its operational semantics and denotational semantics which is based on the Scott-continuous function space model. For ease of exposition, it is good to make a clear distinction between reduction systems and programming languages. A reduction system is a (formal) language with rewrite rules. A programming language may be regarded as a reduction system with some additional features. For our purpose, it is an essential feature of a programming language to be equipped with a specified reduction strategy. In this section, we study the operational properties of the following systems.

---

<sup>7</sup>In [Scott, 1969], Scott considered a version of PCF based on typed S- and K-combinators.

$\lambda_V$	untyped $\lambda$ -calculus with fixed-point operators
$\lambda_V^\rightarrow$	simply typed $\lambda$ -calculus with fixed-point operators
$\lambda_V^\rightarrow(\mathbf{A})$	$\lambda_V^\rightarrow$ augmented by a set $\mathbf{A}$ of basic arithmetic operations.
$\text{app}^\rightarrow(\mathbf{A})$	subcollection of $\lambda_V^\rightarrow(\mathbf{A})$ generated from $\mathbf{A}$ and ground-type $\Omega$ s.

The programming language PCF is obtained from the reduction system  $\lambda_V^\rightarrow(\mathbf{A})$  by giving it a specific reduction strategy. We shall study the operational properties of the reduction system  $\lambda_V^\rightarrow(\mathbf{A})$  so as to understand those of the programming language PCF.

## 2.2 Syntax of the reduction system $\lambda_V^\rightarrow(\mathbf{A})$

**Types** We use meta-variables  $\sigma, \tau, \nu$  to range over *types* which are defined as follows:

$$\begin{array}{ll}
 \sigma ::= \iota & \text{natural numbers} \\
 | \quad o & \text{booleans} \\
 | \quad \sigma \Rightarrow \tau & \text{arrow or function type.}
 \end{array}$$

So the types that we are concerned with are just those that are considered in Church's simple theory of types [Church, 1940]. We use the meta-variable  $\beta$  to range over ground types  $\iota$  and  $o$ . As usual,  $\Rightarrow$  associates to the right:  $\sigma \Rightarrow \tau \Rightarrow \nu$  is read as  $\sigma \Rightarrow (\tau \Rightarrow \nu)$ . Note that with  $n \geq 0$ , each simple type can be uniquely expressed as

$$\sigma_1 \Rightarrow \sigma_2 \cdots \Rightarrow \sigma_n \Rightarrow \beta$$

which we abbreviate as  $(\sigma_1, \sigma_2, \dots, \sigma_n, \beta)$ . The *rank* of a type is defined as follows:

$$\begin{aligned}
 \text{rank}(\beta) &\stackrel{\text{def}}{=} 0, \\
 \text{rank}(\sigma \Rightarrow \tau) &\stackrel{\text{def}}{=} \max(\text{rank}(\sigma) + 1, \text{rank}(\tau)).
 \end{aligned}$$

The rank measures how “higher-order” a type is. A ground type  $\beta$  has rank 0. If  $\text{rank}(\sigma) = 1$ , then  $\sigma$  is the usual first-order type. In general, for  $n \geq 0$ , we call  $\sigma$  an  $n$ -th order type if  $\text{rank}(\sigma) = n$ . Many of the mathematical difficulties associated with higher-order objects (some of which we will encounter in this chapter) stem from nesting on the left of the arrow type.

**Terms** For each type  $\sigma$ , we fix a denumerable set  $\{x^\sigma\}$  of variables. Let  $\mathbf{C} = \{c^\sigma\}$  be a set of typed constants. *Raw expressions* of the language  $\lambda_V^-(\mathbf{C})$  parametrized over the set  $\mathbf{C}$  of constants are defined by the following grammar:

$M ::= \Omega^\sigma$	undefined term
$c^\sigma$	constant
$x^\sigma$	variable
$(M \cdot M)$	application
$(\lambda x^\sigma.M)$	$\lambda$ -abstraction
$\mathbf{Y}^\sigma(M)$	$\mathbf{Y}$ -term.

Whenever type information is irrelevant, we omit type labels and write  $\Omega^\sigma$ ,  $x^\sigma$  and  $\mathbf{Y}^\sigma(-)$  simply as  $\Omega$ ,  $x$  and  $\mathbf{Y}(-)$  respectively. The application  $(M \cdot N)$  is written simply as  $MN$ . As usual, application associates to the left:  $MN_1 \cdots N_n$  abbreviates  $(\cdots ((MN_1)N_2) \cdots N_n)$ , and we routinely omit as many parentheses as we safely can. An *untyped* term is simply a raw expression with type labels erased.

**Typing rules for  $\lambda_V^-(\mathbf{A})$**  Given a collection  $\mathbf{C}$  of typed constants, the reduction system (or term rewriting system)  $\lambda_V^-(\mathbf{C})$  consists of raw expressions which are well-typed. The phrase  $M : \sigma$  is an assertion that the type of the term  $M$  is  $\sigma$ , which is inductively defined according to the following rules:

$$\begin{array}{c}
 x^\sigma : \sigma \qquad \qquad \qquad \Omega^\sigma : \sigma \\
 \\
 c^\sigma : \sigma \qquad \qquad \qquad \frac{M : \sigma \Rightarrow \sigma}{\mathbf{Y}^\sigma(M) : \sigma} \\
 \\
 \frac{M : \sigma \Rightarrow \tau \quad N : \sigma}{(M \cdot N) : \tau} \qquad \qquad \frac{M : \tau}{(\lambda x^\sigma.M) : \sigma \Rightarrow \tau}
 \end{array}$$

We write the collection  $\lambda_V^-(\emptyset)$  of terms simply as  $\lambda_V^-$ .

**Arithmetic operations** We fix a set  $\mathbf{A}$  of basic arithmetic operations. They are listed together with their types as follows:



$n$	: $\iota$	numerals, for each natural number $n \geq 0$
$t, f$	: $o$	booleans: truth and falsity
$\text{succ}$	: $\iota \Rightarrow \iota$	successor constant
$\text{pred}$	: $\iota \Rightarrow \iota$	predecessor constant
$\text{zero?}$	: $\iota \Rightarrow o$	test for zero constant
$\text{cond}^\iota$	: $o \Rightarrow \iota \Rightarrow \iota \Rightarrow \iota$	integer conditional
$\text{cond}^o$	: $o \Rightarrow o \Rightarrow o \Rightarrow o$	boolean conditional.

We define the subcollection  $\text{app}^\rightarrow(\mathbf{A})$  of  $\lambda_{\vec{V}}^\rightarrow(\mathbf{A})$  consisting of applicative  $\lambda_{\vec{V}}^\rightarrow(\mathbf{A})$ -terms generated solely from  $\mathbf{A}$  and the undefined terms  $\Omega$  of ground type. Formally, we define the raw expressions of  $\text{app}^\rightarrow(\mathbf{A})$  as follows: with  $\beta$  ranging over ground types, and  $c$  over  $\mathbf{A}$ ,

$$A ::= \Omega^\beta \mid c$$

and an  $\text{app}^\rightarrow(\mathbf{A})$ -term is a simply typable raw expression. In other words, an  $\text{app}^\rightarrow(\mathbf{A})$ -term is a  $\lambda_{\vec{V}}^\rightarrow(\mathbf{A})$ -term which does not contain any of the following as subterms: variables, abstractions,  $Y$ -terms and higher-order  $\Omega$ s. Note that  $\text{app}^\rightarrow(\mathbf{A})$ -terms are simply typable, by construction. The notion of free and bound variables is completely standard; a *closed* term is a term without any free variables. In this chapter, equivalence  $\equiv$  between terms denotes syntactic equality modulo renaming of bound variables. We write the *term substitution* (as opposed to context substitution) operation as  $M[N/x^\sigma]$  which means “in  $M$ , substitute the term  $N$  for every free occurrence of  $x^\sigma$ ”, taking care to rename bound variables where necessary so as to avoid variable capture. See, for example, [Barendregt, 1984, p. 27] for a formal definition.

*Contexts* are just terms with (typed) “holes” as subterms which may be filled with type-compatible terms. The holes are ranged over by meta-variables  $X_i^\sigma$ ,  $Y_j^\tau$  etc. As before, type labels are omitted from holes  $X^\sigma$  whenever we can get away with it. Raw  $\lambda_{\vec{V}}^\rightarrow(\mathbf{C})$ -contexts are ranged over by meta-variables  $C$ , and are defined as follows:

$$C ::= X^\sigma \mid \Omega^\sigma \mid c^\sigma \mid x \mid (\lambda x^\sigma. C) \mid (C \cdot C) \mid (Y^\sigma(C)).$$

We write a context with holes  $X_1, \dots, X_n$  as  $C[X_1, \dots, X_n]$ . A  $\lambda_{\vec{V}}^\rightarrow(\mathbf{C})$ -context is just a raw  $\lambda_{\vec{V}}^\rightarrow(\mathbf{C})$ -context which is well-typed according to the typing rules for  $\lambda_{\vec{V}}^\rightarrow(\mathbf{C})$ -terms. *Context substitution* is an operation defined recursively as follows: for terms  $P_1, \dots, P_n$  which are type-compatible with holes  $X_1^{\sigma_1}, \dots, X_n^{\sigma_n}$  respectively,  $C[P_1, \dots, P_n]$  is defined to be



$$\left\{ \begin{array}{ll} C & \text{if } C \text{ is a variable} \\ & \text{or constant } j \\ P_i & \text{if } C \text{ is the hole } X_i^{\sigma_i} \\ (C_1[P_1, \dots, P_n] \cdot C_2[P_1, \dots, P_n]) & \text{if } C = (C_1 \cdot C_2) \\ (\lambda x^\sigma. C'[P_1, \dots, P_n]) & \text{if } C = (\lambda x^\sigma. C') \\ (\mathbf{Y}^\sigma(C'[P_1, \dots, P_n])) & \text{if } C = (\mathbf{Y}^\sigma(C')). \end{array} \right.$$

It is important to note the difference between context-substitution and term-substitution: variables may well be bound as a result of a context-substitution. For example, take  $C[X]$  to be  $\lambda x. Xx$ ; then  $C[xy]$  is  $\lambda x. (xy)x$ . By contrast,  $C[X][(xy)/X]$  is  $\lambda z. (xy)z$ ; note that the bound variable  $x$  in  $\lambda x. Xx$  is renamed to  $z$  to avoid capture.

### 2.3 Reduction rules of the system $\lambda_V^{\rightarrow}(\mathbf{A})$

Reduction or rewriting in the system  $\lambda_V^{\rightarrow}(\mathbf{A})$  is represented as a binary relation  $\rightarrow$  over closed terms. We read  $M \rightarrow N$  as “the term  $M$  reduces in one step to  $N$ ”. The reduction system  $\lambda_V^{\rightarrow}(\mathbf{A})$  as presented in the following is non-deterministic: since it is not committed to any reduction strategy, it is not (yet) a programming language. Our purpose in this section is to study those operational properties of  $\lambda_V^{\rightarrow}(\mathbf{A})$  which are derived purely from the syntax of the system and are independent of the reduction strategy. The binary relation  $\rightarrow$  is defined recursively by two sets of axiom and rule schemes as follows:

- **$\lambda$ -reductions:**

$$(\lambda x^\sigma. M)N \rightarrow M[N/x^\sigma] \qquad \mathbf{Y}^\sigma(M) \rightarrow M(\mathbf{Y}^\sigma(M))$$

$$\Omega^\sigma \rightarrow \Omega^\sigma$$

- **$\delta$ -reductions:** we let  $\beta$  range over ground types,

$$\text{cond}^\beta \mathbf{t} \rightarrow \lambda x^\beta. (\lambda y^\beta. x) \qquad \text{cond}^\beta \mathbf{f} \rightarrow \lambda x^\beta. (\lambda y^\beta. y)$$

$$\text{succ } n \rightarrow n + 1 \qquad \text{pred } n + 1 \rightarrow n$$

$$\text{pred } 0 \rightarrow 0 \qquad \text{zero? } n + 1 \rightarrow \mathbf{f}$$

$$\text{zero? } 0 \rightarrow \mathbf{t} \qquad \Omega^\alpha \rightarrow \Omega^\alpha$$

• **compatible closure:**

$$\frac{M \rightarrow M'}{C[M] \rightarrow C[M']}.$$

Note that for ground types  $\alpha$ , the axiom  $\Omega^\beta \rightarrow \Omega^\beta$  is both a  $\lambda$ - as well as a  $\delta$ -rule. The left hand side of an instance of a  $\lambda$ -reduction axiom scheme is called a  $\lambda$ -*redex*; similarly, we define  $\delta$ -*redex*. Reduction “under a lambda” is permissible in the reduction system (but not in the programming language PCF as we shall see later). For example,  $\lambda x^t.\text{succ}1 \rightarrow \lambda x^t 2$  is derivable.

The reduction system  $\rightarrow$  is well-behaved. A number of desirable properties can immediately be observed.

**Lemma 2.3.1 (Unique Decomposition).** *For any  $\lambda_V^{\rightarrow}(\mathbf{A})$ -term  $M$ , whenever  $M \rightarrow M'$ , there is a unique  $\lambda_V^{\rightarrow}(\mathbf{A})$ -context  $C[X]$  such that  $M \equiv C[P]$ ,  $M' \equiv C[Q]$  and that  $P \rightarrow Q$  is an instance of either a  $\lambda$ -reduction axiom scheme or  $\delta$ -reduction. ■*

The result is trivially true. Using the notation of the lemma, whenever  $P \rightarrow Q$  is a  $\lambda$ -reduction, we shall write  $M \rightarrow_\lambda M'$ ; otherwise if  $P \rightarrow Q$  is a  $\delta$ -reduction, we shall write  $M \rightarrow_\delta M'$ . If no subterm of  $M$  is a  $\lambda$ -redex then we call  $M$  a  $\lambda$ -normal form or  $\lambda$ -nf for short; similarly, we define a  $\delta$ -normal form. A term  $M$  is a  $\rightarrow$ -normal form or simply a normal form if there is no term  $N$  such that  $M \rightarrow N$  holds. It is easy to see that the type of a term is preserved by reduction. More precisely:

**Lemma 2.3.2 (Subject Reduction).** *For any  $\lambda_V^{\rightarrow}(\mathbf{A})$ -term  $M : \sigma$ , whenever  $M \rightarrow N$ , then  $N : \sigma$ . ■*

A  $\lambda_V^{\rightarrow}(\mathbf{A})$ -*program* (or simply a program) is a closed  $\lambda_V^{\rightarrow}(\mathbf{A})$ -term which is of ground type. *Ground values* are the numerals and booleans. We let meta-variable  $V$  range over ground values. By a simple syntactic case analysis, if a program  $M$  is in  $\rightarrow$ -normal form, then it is a ground value. This means that whenever the computation of a program terminates, the outcome is guaranteed to be a meaningful piece of data, rather than some unintended piece of code like  $\Omega'$ . For this reason, we admit the reduction  $\Omega^\sigma \rightarrow \Omega^\sigma$ .

## 2.4 Operational properties of $\lambda_V^{\rightarrow}(\mathbf{A})$

We define  $\twoheadrightarrow$  to be the reflexive, transitive closure of  $\rightarrow$ . Our goal is to study the operational properties of the reduction system  $(\lambda_V^{\rightarrow}(\mathbf{A}), \twoheadrightarrow)$ . The four major properties of the system are:

- Church-Rosser property,
- Standardization Theorem,
- sequentiality,

• stability.

The proof for the Church-Rosser property of  $\lambda_V^-(\mathbf{A})$  may be organized into the following stages. First, recall that a binary relation  $\mathbf{R}$  over a set  $S$  is said to satisfy the *diamond property* if whenever  $s \mathbf{R} s_1$  and  $s \mathbf{R} s_2$ , then there is some  $t$  such that  $s_1 \mathbf{R} t$  and  $s_2 \mathbf{R} t$ .

**Lemma 2.4.1.**

- (i) The binary relation  $\rightarrow_\lambda$  over  $\lambda_V^-(\mathbf{A})$  satisfies the diamond property.
- (ii) The binary relation  $\rightarrow_\delta$  over  $\lambda_V^-(\mathbf{A})$  satisfies the diamond property.
- (iii) The binary relations  $\rightarrow_\lambda$  and  $\rightarrow_\delta$  commute; that is to say, whenever  $M \rightarrow_\lambda M_1$  and  $M \rightarrow_\delta M_2$ , then there is some  $N$  such that  $M_1 \rightarrow_\delta N$  and  $M_2 \rightarrow_\lambda N$ . ■

Part (i) of the lemma is a consequence of the following: whenever  $M \rightarrow_\lambda M_1$  and  $M \rightarrow_\lambda M_2$ , then there is some  $N$  such that  $M_1 \rightarrow_\lambda N$  and that  $M_2 \rightarrow_\lambda N$ ; which may be proved by a tedious syntactic case analysis. Parts (ii) and (iii) may be easily established by a similar case analysis.

Using the three results above, and by an appeal to a lemma<sup>8</sup> of Hindley and Rosen (see, for example, [Barendregt, 1984, p. 64]), we get:

**Proposition 2.4.2 (Church-Rosser).** *The reduction  $\rightarrow$  over  $\lambda_V^-(\mathbf{A})$  is Church-Rosser, i.e. whenever  $M \twoheadrightarrow M_1$  and  $M \twoheadrightarrow M_2$ , then there is some  $N$  such that  $M_1 \twoheadrightarrow N$  and  $M_2 \twoheadrightarrow N$ .* ■

**Separation argument** The Church-Rosser property of  $\lambda_V^-(\mathbf{A})$  has been established by first considering the Church-Rosser property for  $\lambda$ -reductions and  $\delta$ -reductions separately. The argument is then completed by appealing to the Hindley-Rosen Lemma. When we establish the property of sequentiality and stability, a somewhat more sophisticated kind of separation argument will be used. First, we establish the following by a straightforward case analysis:

**Lemma 2.4.3.** *For any  $\lambda_V^-(\mathbf{A})$ -term  $M$ , if  $M \rightarrow_\delta P \rightarrow_\lambda N$ , then there is some  $Q$  such that  $M \rightarrow_\lambda Q \twoheadrightarrow_\delta N$ .* ■

Given the lemma, it is then a matter of a simple inductive argument to establish the following preliminary separation result.

**Lemma 2.4.4.** *For any  $\lambda_V^-(\mathbf{A})$ -term  $M$ , if  $M \twoheadrightarrow N$ , then there is some  $P$  such that  $M \twoheadrightarrow_\lambda P \twoheadrightarrow_\delta N$ .* ■

**Example 2.4.5.** We define “left-add”  $\mathbf{l-add} \equiv \mathbf{Y}(F)$  where

$$F \equiv \lambda f. \lambda mn. \text{cond}^u(\text{zero?}m)n(f(\text{pred}m)(\text{succ}n))$$

<sup>8</sup>The Hindley-Rosen Lemma states that if binary relations  $\mathbf{R}_1$  and  $\mathbf{R}_2$  over a set  $S$  both satisfy the diamond property, and that  $\mathbf{R}_1$  commutes with  $\mathbf{R}_2$ , then the reflexive, transitive closure of  $\mathbf{R}_1 \cup \mathbf{R}_2$  satisfies the diamond property.

(we omit type labels). Consider, for example, the reduction:

$$\text{l-add succ0}((\lambda y.y)2) \rightarrow 3$$

following any reduction strategy we care to choose. We can organize the computation of the program in the following way:

$$\begin{aligned} & \text{l-add}(\text{succ0})((\lambda y.y)2) \\ \rightarrow_{\lambda} & \text{l-add}(\text{succ0})2 \\ \rightarrow_{\lambda} & \text{cond}^t(\text{zero?}(\text{succ0}))2(\text{cond}^t(\text{zero?}(\text{pred}(\text{succ0}))) \\ & (\text{succ2})(\mathbf{Y}(F)(\text{pred}(\text{pred}(\text{succ0}))) (\text{succ}(\text{succ2})))) \\ \rightarrow_{\delta} & \text{cond}^t(\text{zero?}(\text{pred}(\text{succ0}))) \\ & (\text{succ2})(\mathbf{Y}(F)(\text{pred}(\text{pred}(\text{succ0}))) (\text{succ}(\text{succ2})))) \\ \rightarrow_{\delta} & 3. \end{aligned}$$

The preceding lemma shows that an arbitrary reduction is “equivalent” to a reduction which is partitioned into  $\lambda$ -reductions and  $\delta$ -reductions: the latter have been postponed to the end.  $\lambda$ -reductions pertain to the higher-order part of the computation, whereas  $\delta$ -reductions perform essentially first-order computations. A clean separation of the two kinds of reductions is desirable: techniques tailored for one may be applied in isolation from the other to greater effect. For example, the following are intuitively clear:

- Properties of  $\lambda$ -reductions in  $\langle \lambda_{\vec{Y}}^{\rightarrow}(\mathbf{A}), \rightarrow \rangle$  are essentially those of  $\langle \lambda_{\vec{Y}}^{\rightarrow}(\mathbf{A}), \rightarrow_{\lambda} \rangle$ .
- It suffices to study  $\delta$ -reductions by focusing on terms of the first-order applicative algebra  $\text{app}^{\rightarrow}(\mathbf{A})$ .

However, the separation spelt out in the preceding lemma is not as sharp as it can be. Suppose  $M$  in the preceding proposition is a program and  $N$  is a ground value; there is no guarantee that the intermediate term  $P$  will always belong to  $\text{app}^{\rightarrow}(\mathbf{A})$ , since  $P$  may well contain a first-order  $\lambda$ -redex. This is the case, for instance, in Example 2.4.5, where an intermediate term in the reduction has the  $\lambda$ -redex  $\mathbf{Y}(F)$  as a subterm.

To get a sharper result, we use the notion of approximate term due to Wadsworth [Wadsworth, 1978]. Suppose we start reducing a term  $M$  following any reduction strategy we care to choose. If  $M$  has a normal form (then the Church-Rosser property assures us that the normal form is unique), and the reduction actually reaches it, then we may quite reasonably take the normal form as the value of the reduction. If the reduction should terminate at a term  $M'$  which is not a normal form, what should the value of the reduction be? It is possible that  $M'$  has

no normal form; in which case, one could assign the “value”  $\Omega$  to such a reduction. But this would be too drastic. Consider the case of  $M \equiv (\lambda x.x)y(\lambda uv.v)((\lambda y.y)(\lambda y.y))(\mathbf{Y}(\lambda z.z))$ , and the reduction

$$M \twoheadrightarrow y(\lambda uv.v)((\lambda y.y)(\lambda y.y))(\mathbf{Y}(\lambda z.z)) \equiv N.$$

The term  $N$  is not in normal form but enough  $\lambda$ -reduction has already taken place for us to conclude that *were*  $M$  to have a normal form, then it would have to fit the syntactic shape of  $y(\lambda uv.v)\Omega\Omega$  where  $\Omega$  here stands for an indeterminate subterm. Further reduction starting from  $N$  may reveal more information about the subterms represented by  $\Omega$ . The term  $y(\lambda uv.v)\Omega\Omega$  is called the  $\lambda$ -approximant of  $N$ , which is written as  $\omega(N)$ , because we replace all  $\lambda$ -redexes by  $\Omega$ . Generally, given a  $\lambda_{\vec{v}}^-(\mathbf{A})$ -term  $M$ , we define the  $\lambda$ -approximant  $\omega(M)$  of  $M$  recursively as: for  $n \geq 0$ , and with  $\xi$  ranging over variables, and constants from the set  $\mathbf{A}$ ,

$$\begin{aligned} \omega(\lambda \vec{x}.\Omega \vec{M}) &\stackrel{\text{def}}{=} \Omega, \\ \omega(\lambda \vec{x}.(\lambda x.P)Q \vec{M}) &\stackrel{\text{def}}{=} \Omega, \\ \omega(\lambda \vec{x}.\mathbf{Y}(P) \vec{M}) &\stackrel{\text{def}}{=} \Omega, \\ \omega(\lambda \vec{x}.\xi M_1 \cdots M_n) &\stackrel{\text{def}}{=} \lambda \vec{x}.\xi \omega(M_1) \cdots \omega(M_n). \end{aligned}$$

For example,  $\omega(ysucc((\lambda xy.x)12)) = ysucc\Omega$ . Note that it is always the case that  $\omega(M)$  matches  $M$  exactly, except at occurrences of  $\Omega$  in  $\omega(M)$ . We write this well-known “ $\Omega$ -match” partial order as  $\leq_{\Omega}$ . Formally, the relation  $\leq_{\Omega}$  over  $\lambda_{\vec{v}}^-(\mathbf{A})$ -terms may be defined by the axiom scheme

$$C[\Omega, \dots, \Omega] \leq_{\Omega} C[M_1, \dots, M_n]$$

where  $n \geq 0$ , and  $C[X_1, \dots, X_n]$  ranges over all  $\lambda_{\vec{v}}^-(\mathbf{A})$ -contexts, and  $M_i$  over  $\lambda_{\vec{v}}^-(\mathbf{A})$ -terms. Clearly,  $\omega(-)$  is idempotent. As an easy exercise, the reader may wish to verify the following properties: for any  $\lambda_{\vec{v}}^-(\mathbf{A})$ -terms  $M, N$ ,

- (i) If  $M \leq_{\Omega} N$ , then  $\omega(M) \leq_{\Omega} \omega(N)$ .
- (ii) If  $M \rightarrow N$ , then  $\omega(M) \leq_{\Omega} \omega(N)$ .
- (iii) For any program  $M$ , and with  $V$  ranging over ground values, if  $M \rightarrow V$ , then there is a program  $N$  such that  $M \twoheadrightarrow_{\lambda} N$  and  $\omega(N) \twoheadrightarrow_{\delta} V$ .

**Lemma 2.4.6.** *Given any  $A, A' \in \text{app}^-(\mathbf{A})$  such that  $A \leq_{\Omega} A'$ , if  $A \twoheadrightarrow_{\delta} V$  for some ground value  $V$ , then  $A' \twoheadrightarrow_{\delta} V$ .  $\blacksquare$*

A complete division of the reduction of programs into

- higher-order reduction in the reduction system  $\langle \lambda_{\vec{v}}^-(\mathbf{A}), \rightarrow_{\lambda} \rangle$ , and
- first-order reduction in the reduction system  $\langle \text{app}^-(\mathbf{A}), \rightarrow_{\delta} \rangle$



is now possible as follows:

**Proposition 2.4.7 (Separation).** *For any program  $M$ , and with  $V$  ranging over ground values, then  $M \rightarrow V$  if and only if there is a program  $N$  such that  $M \rightarrow_\lambda N$  and  $\omega(N) \rightarrow_\delta V$ .* ■

An important point of the separation property is this: there is no loss of generality in studying properties of  $\delta$ -reductions restricted to terms from the first-order algebra  $\text{app}^\rightarrow(\mathbf{A})$ . This proposition may be proved by a straightforward structural induction on  $\omega(M)$ .

If there is a computation of a program  $M$  which terminates at a normal form, we have already observed that the normal form is a ground value. By the Church-Rosser property, every terminating computation of the program  $M$  produces the same ground value. Further, the Separation Proposition asserts that there is a particular terminating computation of  $M$  which first performs  $\lambda$ -reduction, and then performs  $\delta$ -reduction on terms of the first-order applicative algebra  $\text{app}^\rightarrow(\mathbf{A})$ .

## 2.5 Böhm trees and the Syntactic Continuity Theorem

The abstract syntax of a term  $M$  is best described as a tree  $T_M$ ; subterms of  $M$  then correspond to subtrees of the tree  $T_M$ . Any subterm  $N$  of  $M$  may be precisely identified by tracing the uniquely defined path from the root of the tree  $T_M$  to the root of the corresponding subtree  $T_N$ . In term rewriting, such a path is called the *occurrence* of  $N$  in  $M$ . Abstractly, an occurrence is just a finite sequence of elements from the set  $\{0, 1, 2\}$ . The set  $\{0, 1, 2\}^*$  of occurrences is ranged over by meta-variables  $\alpha, \beta$ . We read  $M / \alpha$  as “the subterm of  $M$  occurring at  $\alpha$ ”. Such a subterm may or may not be defined. The concatenation of sequences  $\alpha$  and  $\beta$  is written as  $\alpha \cdot \beta$ . The prefix ordering  $\leq$  is defined in the standard way: we say that  $\alpha \leq \beta$  if and only if  $\beta = \alpha \cdot \gamma$  for some sequence  $\gamma$ . The empty sequence is written as  $\epsilon$ .

Formally, with  $M, N$  ranging over  $\lambda_Y^-(\mathbf{A})$ -terms and  $\alpha$  over occurrences, we read  $M / \alpha = N$  as a 3-tuple  $\langle M, \alpha, N \rangle$ , and define it as a relation, i.e. a subset of  $\lambda_Y^-(\mathbf{A}) \times \{0, 1, 2\}^* \times \lambda_Y^-(\mathbf{A})$ . The reader should have no difficulty in showing that the relation is actually a partial function mapping  $\langle M, \alpha \rangle$  from  $\lambda_Y^-(\mathbf{A}) \times \{0, 1, 2\}^*$  to an element  $M / \alpha$  of  $\lambda_Y^-(\mathbf{A})$ ;  $M / \alpha$  is defined and equal to  $N$  whenever  $M / \alpha = N$  is derivable. For  $i = 1, 2$ ,

$$\begin{array}{c}
 M / \epsilon = M \\
 \hline
 M / \alpha = N \\
 \hline
 (\lambda x^\sigma.M) / 0 \cdot \alpha = N
 \end{array}
 \qquad
 \begin{array}{c}
 M_i / \alpha = N \\
 \hline
 (M_1 \cdot M_2) / i \cdot \alpha = N \\
 \hline
 M / \alpha = N \\
 \hline
 (\mathbf{Y}^\sigma(M)) / 0 \cdot \alpha = N.
 \end{array}$$

For example, consider  $M \equiv (\lambda y.y(\text{succ}1))(\mathbf{Y}((\lambda x.x)(\lambda x.\bar{x})))$ . The subterm

$\bar{x}$  of  $M$  occurs at  $2 \cdot 0 \cdot 2 \cdot 0$ , and  $M / 1 \cdot 0 \cdot 2 \cdot 2 = 1$ ; while  $M / 1 \cdot 0 \cdot 2 \cdot 0$  is undefined.

**Approximate normal forms** The collection of *partial normal forms* of the language  $\lambda_{\vec{v}}(\mathbf{A})$  is defined as follows. We let  $A, A_i$  range over partial normal forms, and  $\xi$  over variables, and constants from the set  $\mathbf{A}$ . For any  $m, n \geq 0$ ,

$$A ::= \Omega \mid \lambda x_1 \cdots x_n. \xi A_1 \cdots A_m.$$

For example,  $\lambda xy.x\Omega$ ,  $x\text{cond}^t\text{f}(\text{succ}z)(y\Omega)$  are partial normal forms; but

$$\lambda x.\Omega, \quad \mathbf{Y}(\lambda x.yx) \quad \text{and} \quad \Omega(\text{succ}2)((\lambda y.y)0)$$

are not. Incidentally, partial normal forms are *not* the same as  $\lambda$ -normal forms: rather, they are precisely the image of the map

$$\omega(-) : \lambda_{\vec{v}}(\mathbf{A}) \rightarrow \lambda_{\vec{v}}(\mathbf{A}).$$

For this reason, we write  $\omega(\lambda_{\vec{v}}(\mathbf{A}))$  as the collection of all partial normal forms.

For any  $\lambda_{\vec{v}}(\mathbf{A})$ -term  $M$ , an *approximate normal form* of  $M$  is a partial normal form which  $\leq_n$ -approximates  $M$ . We define the collection  $\text{ANF}(M)$  of approximate normal forms of  $M$  as

$$\text{ANF}(M) \stackrel{\text{def}}{=} \{ A \in \omega(\lambda_{\vec{v}}(\mathbf{A})) : A \leq_n M \}.$$

It is easy to see that for any  $A \in \text{ANF}(M)$ ,  $A \leq_n \omega(M)$ .

**Böhm trees** In the case of a program  $M$  in the reduction system  $\langle \lambda_{\vec{v}}(\mathbf{A}), \rightarrow \rangle$ , its value, if it exists, is naturally that unique ground value to which  $M$  reduces. In the case of an arbitrary  $\lambda_{\vec{v}}(\mathbf{A})$ -term  $M$  in the reduction system  $\langle \lambda_{\vec{v}}(\mathbf{A}), \rightarrow_{\lambda} \rangle$  (in which no  $\delta$ -reductions are allowed), what should its “value” be?

The answer, following standard practice in sensible<sup>9</sup>  $\lambda$ -calculus, lies in the notion of *Böhm tree* [Böhm, 1968]. For any PCF-term  $M$ , the *Böhm tree*  $\text{BT}(M)$  of  $M$  is defined as

$$\begin{aligned} \text{BT}(M) &\stackrel{\text{def}}{=} \{ A \in \text{ANF}(N) : M \rightarrow_{\lambda} N \} \\ &= \{ A \in \omega(\lambda_{\vec{v}}(\mathbf{A})) : A \leq_n N \ \& \ M \rightarrow_{\lambda} N \}. \end{aligned}$$

We claim that for any PCF-term  $M$ ,  $\text{BT}(M)$  is an ideal of the partially-ordered set  $\langle \omega(\lambda_{\vec{v}}(\mathbf{A})), \leq_n \rangle$ . That is to say,

<sup>9</sup>We use the adjective *sensible* in the technical sense of [Barendregt, 1984]. In the sensible setting, a  $\lambda$ -term is meaningful if and only if it has a head normal form.

- (1)  $\text{BT}(M)$  is a  $\leq_\Omega$ -down-closed subset of  $\langle \omega(\lambda_{\vec{v}}(\mathbf{A})), \leq_\Omega \rangle$ ;
- (2) for any  $A_1, A_2 \in \text{BT}(M)$ , there is some  $A \in \text{BT}(M)$  such that both  $A_1, A_2 \leq_\Omega A$ , i.e.  $\text{BT}(M)$  is a  $\leq_\Omega$ -directed subset of  $\langle \omega(\lambda_{\vec{v}}(\mathbf{A})), \leq_\Omega \rangle$ .

(1) follows immediately from the definition. To see (2), suppose  $A_i \in \text{ANF}(N_i)$  with  $M \rightarrow N_i$ , for  $i = 1, 2$ . By the Church-Rosser property, there is some  $N$  such that  $N_i \rightarrow N$  for both  $i = 1$  and  $2$ . Since  $\omega(N_i) \leq_\Omega \omega(N)$ , we have  $A_i \leq_\Omega \omega(N)$  for both  $i = 1$  and  $2$ .

It is a well-known result (see e.g. [Gunter, 1992, Ch. 5]) that given any preorder  $\langle X, \leq \rangle$ , the collection  $\text{Idl}(X)$  of ideals ordered by subset inclusion is an algebraic CPO. Further,

- The compact elements of  $\text{Idl}(X)$  are just the principal ideals of  $\langle X, \leq \rangle$ , i.e. ideals of the form  $\downarrow y = \{x : x \leq y\}$ .
- There is a canonical embedding  $I$  of  $\langle X, \leq \rangle$  into  $\text{Idl}(X)$  which maps an element  $x$  of  $X$  to the principal ideal  $\downarrow x$  such that for any CPO  $Y$  and any monotonic function  $f : X \rightarrow Y$ , there is a unique continuous function  $\bar{f} : \text{Idl}(X) \rightarrow Y$  extending  $f$ , i.e.  $\bar{f} \circ I = f$ .

Formally, we represent a Böhm tree  $\text{BT}(M)$  as the corresponding element of  $\text{Idl}(\langle \omega(\lambda_{\vec{v}}(\mathbf{A})), \leq_\Omega \rangle)$ . Whenever  $\text{BT}(M)$  is a compact element, we confuse the principal ideal with the  $\leq_\Omega$ -maximal element of the ideal. For example, given  $M \equiv (\lambda xy.x)1(\text{zero}?y)((\lambda xy.yx)(\text{succ}0))(\mathbf{Y}(\lambda z.z))$ , we have  $\text{BT}(M) \equiv x(\text{zero}?y)(\lambda y.y(\text{succ}0))\Omega$ . We define  $\text{app}^-(\mathbf{A})^\infty$  to be  $\text{Idl}(\langle \text{app}^-(\mathbf{A}), \leq_\Omega \rangle)$ .

**Lemma 2.5.1.** *The following properties of Böhm trees are valid:*

- (i) Whenever  $M \leq_\Omega N$ , then  $\text{BT}(M) \subseteq \text{BT}(N)$ .
- (ii) Whenever  $M \rightarrow_\lambda N$ , then  $\text{BT}(M) = \text{BT}(N)$ .
- (iii) For any program  $P$ ,  $\text{BT}(P) \in \text{app}^-(\mathbf{A})^\infty$ . Further,  $P$  has a  $\rightarrow$ -normal form if and only if  $\text{BT}(P)$  has a  $\delta$ -normal form. ■

An example of an “infinite” element of  $\text{app}^-(\mathbf{A})^\infty$  is  $\Xi 0$  where

$$\Xi \equiv \mathbf{Y}(\lambda f.\lambda n.\text{succ}(fn));$$

so  $\Xi 0$  is the infinite term  $\text{succ}(\text{succ}(\text{succ} \dots))$ . Our formulation of Böhm trees (even when restricted to  $\lambda_{\vec{v}}$ -terms) is different from Barendregt’s in [Barendregt, 1984], though the underlying idea is of course identical. Barendregt represents the Böhm tree of a  $\lambda$ -term as a (possibly infinite) tree; formally, it is a map from occurrences to the set  $\{\lambda \vec{x}.y\} \cup \{\perp\}$  of labels, subject to the usual structural requirements of a tree plus the condition that  $\perp$  can only occur at terminal nodes. A difference to note is the notion of a subtree of a Böhm tree. For any  $\alpha \in \{0, 1, 2\}^*$ , we define the subtree of  $\text{BT}(M)$  occurring at  $\alpha$  as follows:

$$\text{BT}(M) / \alpha \stackrel{\text{def}}{=} \begin{cases} \{ X / \alpha : X \in \text{BT}(M) \} & \text{if } X / \alpha \text{ is defined for some} \\ & X \in \text{BT}(M) \\ \text{undefined} & \text{otherwise.} \end{cases}$$

An important property of a Böhm tree is the following *continuity* result. We direct the reader to [Barendregt, 1984] for a proof.

**Theorem 2.5.2 (Continuity).** *For any  $\lambda_V^-(\mathbf{A})$ -context  $C[X]$ , and any  $\lambda_V^-(\mathbf{A})$ -term  $M$ , then*

$$\text{BT}(C[M]) = \bigcup_{T \leq_\Omega M} \text{BT}(C[T]).$$

■

**Theorem 2.5.3.** *For any program context  $P[X]$ , for any  $\lambda_V^-(\mathbf{A})$ -term  $M$ , and for any ground value  $V$ , if  $P[M] \rightarrow V$  for some ground value  $V$ , then for any  $\lambda_V^-(\mathbf{A})$ -term  $N$  such that  $M \leq_\Omega N$ ,  $P[N] \rightarrow V$ .*

■

## 2.6 Sequentiality and stability

Our task now is to show that programs of the reduction system  $\lambda_V^-(\mathbf{A})$  are sequential and stable. Of course, we have not defined what we mean by sequentiality or stability yet. While the idea of a sequential computation is intuitively clear, and programmers recognize a sequential program when they see one, it is a challenging problem to characterize sequentiality in abstract, mathematical terms. We begin by noting three features of (Kahn-Plotkin) sequentiality:

- *Strictness.* As a first approximation, sequentiality may be observed by strictness in an argument position.
- *State.* Sequentiality is an intensional notion; we need a notion of state (of a computation) in order to express it.
- *Needed channel.* At each such intermediate state, there is a particular input channel into which more input information must be fed in order that further progress in output may be observed.

Consider the conditional construct  $\text{cond}^\circ$  in  $\lambda_V^-(\mathbf{A})$ . We shall think of  $\text{cond}^\circ$  in its “uncurried form” as a three-argument function from  $\mathbb{B}_\perp \times \mathbb{B}_\perp \times \mathbb{B}_\perp$  to  $\mathbb{B}_\perp$  where  $\mathbb{B}_\perp$  is the three-element set  $\{\perp, \text{t}, \text{f}\}$ , and where  $\perp$  is the undefined element. The reader should have no difficulty in convincing himself that  $\text{cond}^\circ$  is a sequential object:  $\text{cond}^\circ$  begins by computing the first argument; hence,  $\text{cond}^\circ$  is strict in the first argument, *i.e.* informally we have  $\text{cond}^\circ(\perp, M, N) = \perp$ , regardless of  $M$  and  $N$ . If the first argument should evaluate to  $\text{t}$ , then  $\text{cond}^\circ(\text{t}, M, N)$  proceeds by evaluating the second argument  $M$ . For this reason,  $\text{cond}^\circ$  is strict in the second (respectively third) argument whenever the first argument is  $\text{t}$  (respectively  $\text{f}$ ).



Intuitively, the meaning of sequentiality of the reduction system  $\langle \lambda_V^-(\mathbf{A}), \rightarrow_\lambda \rangle$  is this: for every  $\lambda_V^-(\mathbf{A})$ -term  $M$ , we think of the occurrences of  $\Omega$  in  $M$  as input channels, and occurrences of  $\Omega$  in  $\text{BT}(M)$  as output channels. The observer monitors progress of a computation at a chosen output channel relative to input to the system fed through the input channels. For each output channel  $\alpha$  (i.e. for every occurrence  $\alpha$  of  $\Omega$  in  $\text{BT}(M)$ ),

- *either* no additional amount of input fed into any of the input channels will bring about any observable increase at the output channel  $\alpha$  (i.e.  $\Omega$  occurring at  $\alpha$  in  $\text{BT}(M)$  is “caused” by some subterm of  $M$  which is not any of the occurrences of  $\Omega$  in  $M$ );
- *or* there is a unique input channel  $\beta$  such that any additional information observed at the output channel  $\alpha$  necessarily comes from additional information fed into the unique input channel  $\beta$  (i.e.  $\Omega$  occurring at  $\alpha$  in  $\text{BT}(M)$  at  $\alpha$  is “caused” by a unique occurrence of  $\Omega$  in  $M$  at  $\beta$ ).

**Theorem 2.6.1 (Sequentiality of  $\langle \lambda_V^-(\mathbf{A}), \rightarrow_\lambda \rangle$ ).** *For any  $\lambda_V^-(\mathbf{A})$ -term  $M$ , and for any occurrence  $\alpha$  such that  $\text{BT}(M) / \alpha = \Omega$ , exactly one of the following is valid:*

- (i) *for any  $\lambda_V^-(\mathbf{A})$ -term  $N$ , if  $M \leq_\Omega N$  then  $\text{BT}(N) / \alpha = \Omega$ ,*
- (ii) *there is a unique occurrence  $\beta$  of  $\Omega$  in  $M$  such that for any  $\lambda_V^-(\mathbf{A})$ -term  $N$ ,*

$$M \leq_\Omega N \ \& \ \text{BT}(N) / \alpha \neq \Omega \quad \implies \quad N / \beta \neq \Omega.$$

*We call  $\beta$  a sequentiality index of  $\text{BT}(-)$  for  $M$  at the occurrence  $\alpha$ .*

**Proof.** We prove by induction on the structure of  $M$ . Given any occurrence  $\alpha$  such that  $\text{BT}(M) / \alpha = \Omega$  (for convenience, we mark this particular occurrence of  $\Omega$  as  $\underline{\Omega}$ ), suppose (i) is not valid. That is to say, there is some  $N$  such that  $M \leq_\Omega N$  and that  $\text{BT}(N) / \alpha \neq \Omega$ . Our intention is to prove (ii). W.l.o.g. we write  $M \equiv \lambda x_1 \cdots x_n. \xi M_1 \cdots M_m$  for  $m \geq 0$ , and so, by supposition,  $N \equiv \lambda x_1 \cdots x_n. \eta N_1 \cdots N_m$  satisfying  $M_i \leq_\Omega N_i$  for each  $i$ . We consider the various cases of  $\xi$  in the following:

CASE  $\xi \equiv \Omega$ : We immediately have  $\text{BT}(M) = \Omega$  and  $\alpha = \epsilon$ . Clearly, (ii) holds, and the sequentiality index  $\beta$  is just the occurrence  $0^n 1^m$  of  $\xi \equiv \Omega$  in  $M$ .

CASE  $\xi \equiv x$  or  $c \in \mathbf{A}$ : By considering the definition of  $\text{BT}(M)$ , we can write  $\alpha$  uniquely as  $\alpha = \alpha_1 \cdot \alpha_2$  such that  $\text{BT}(M_i) / \alpha_2 = \underline{\Omega}$ . Applying the induction hypothesis to the occurrence  $\alpha_2$  of  $\underline{\Omega}$  in  $M_i$ , we can say that exactly one of the following is valid:

- (a) For any term  $U$ , whenever  $M_i \leq_\Omega U$ , then  $\text{BT}(U) / \alpha_2 = \Omega$ . Hence, for any  $N$  such that  $M \leq_\Omega N$ , which implies that  $N \equiv \lambda x_1 \cdots x_n. x N_1 \cdots$



$N_m$ , and in particular that  $M_i \leq_\Omega N_i$ , and so  $\text{BT}(N_i) / \alpha_2 = \Omega$ , which means that  $\text{BT}(N) / \alpha = \Omega$ . But this contradicts our supposition. Hence (a) is impossible.

- (b) There is a unique occurrence  $\beta_0$  such that for any  $U$  satisfying  $M_i \leq_\Omega U$ , then

$$\text{BT}(U) / \alpha_2 \neq \Omega \implies U / \beta_0 \neq \Omega.$$

Now, suppose  $M \leq_\Omega N$  and  $\text{BT}(N) / \alpha \neq \Omega$ . We then have  $M_i \leq_\Omega N_i$  and  $\text{BT}(N_i) / \alpha_2 \neq \Omega$ . And so,  $N_i / \beta_0 \neq \Omega$  which means that  $N / \alpha_1 \cdot \beta_0 \neq \Omega$ . The unique sequentiality index in this case is  $\alpha_1 \cdot \beta_0$ .

CASE  $\xi \equiv \mathbf{Y}^\sigma(L)$ : We first observe the following fact:

Given a  $\lambda_{\mathbf{Y}}^-(\mathbf{A})$ -term  $M$ , whenever  $M \rightarrow_\lambda M'$  and  $M \leq_\Omega N$ , then there is a map  $\theta$  from the set of occurrences of  $\Omega$  in  $M'$  to the set of occurrences of  $\Omega$  in  $M$  such that

$$N \rightarrow_\lambda N \equiv M'[\alpha_1 \mapsto N/\theta(\alpha_1), \dots, \alpha_n \mapsto N/\theta(\alpha_n)];$$

where  $\{\alpha_1, \dots, \alpha_n\} = \{\alpha : M' / \alpha = \Omega\}$ .

There are two cases:

- $\alpha \neq \epsilon$ : We necessarily have  $\text{BT}(M) \neq \Omega$ , and that there is some  $M'$  such that  $M \rightarrow_\lambda M'$  and that  $M' \equiv \lambda x_1 \dots x_p. \xi M_1' \dots M_q'$  with  $\xi$  ranging over variables and  $\mathbf{A}$ . Note that  $\text{BT}(M) = \text{BT}(M')$ , and so  $\text{BT}(M') / \alpha = \Omega$ . Applying the preceding case of  $\xi \equiv x$  to  $M'$ , there is a sequentiality index  $\beta_0$  of  $\text{BT}(-)$  for  $M'$  at the occurrence  $\alpha$ . By the preceding fact, we have  $\text{BT}(N) = \text{BT}(N')$ , and  $M' \leq_\Omega N'$ . By assumption  $\text{BT}(N) / \alpha \neq \Omega$ , so  $\text{BT}(N') / \alpha \neq \Omega$ . Since  $\beta_0$  is a sequentiality index of  $\text{BT}(-)$  for  $M'$  at  $\alpha$ , we have  $N' / \beta_0 \neq \Omega$ . But  $N' / \beta_0 = N / \theta(\beta_0)$ , and so  $N / \theta(\beta_0) \neq \Omega$ , and we are done. The sequentiality index  $\beta$  is  $\theta(\beta_0)$ .
- $\alpha = \epsilon$ : We have  $\text{BT}(M) = \Omega$ . By the Standardization Theorem,  $\text{BT}(M) = \Omega$  if and only if the normal order reduction sequence

$$M \rightarrow_\lambda M^1 \rightarrow_\lambda M^2 \rightarrow_\lambda \dots \rightarrow_\lambda M^n \rightarrow_\lambda \dots$$

satisfies the condition that  $\omega(M^n) = \Omega$  for all  $n$ . There are two cases to consider.

- \* There is some  $i \in \omega$  such that  $M^i \equiv \lambda \vec{x}_1 \dots x_m. \Omega M_1^i \dots M_{k_i}^i$ . In this case, the index is  $0^m 1^{k_i}$ .
- \* For each  $i$ , either  $M^i \equiv \lambda \vec{x}. (\lambda x. P) Q M_1^i \dots M_{k_i}^i$  or  $M^i \equiv \lambda \vec{x}. \mathbf{Y}(P) M_1^i \dots M_{k_i}^i$ . The derivation  $N \rightarrow_\lambda N^1 \rightarrow_\lambda \dots$  defined according to the preceding fact follows the normal order, and so either  $N^i \equiv \lambda \vec{x}. (\lambda x. U) V N_1^i \dots N_{k_i}^i$  or  $N^i \equiv \lambda \vec{x}. \mathbf{Y}(U) N_1^i \dots N_{k_i}^i$ .

Hence  $\text{BT}(N) = \Omega$ , but this contradicts our supposition, and so this subcase is ruled out.

CASE  $\xi \equiv (\lambda x.A)B$ : Exactly the same as the preceding case. ■

This proof is due to Curien [Curien, 1993a, Ch. 2]. Barendregt's book [Barendregt, 1984] also has a treatment of the sequentiality and stability of the pure untyped lambda calculus, though the formulation and proof are rather different.

**Proposition 2.6.2.** *For any program  $M \in \langle \text{app}^\rightarrow(\mathbf{A}), \rightarrow_\delta \rangle$  such that  $M$  has no  $\delta$ -normal form, exactly one of the following is valid:*

- (i) *For any program  $N \in \text{app}^\rightarrow(\mathbf{A})$ , whenever  $M \leq_\Omega N$ , then  $N$  also has no  $\delta$ -normal form.*
- (ii) *There is a unique occurrence  $\beta$  of  $\Omega$  in  $M$  such that for any program  $N \in \text{app}^\rightarrow(\mathbf{A})$ ,*

$$M \leq_\Omega N \ \& \ N \text{ has a } \rightarrow_\delta\text{-normal form} \quad \implies \quad N / \beta \neq \Omega.$$

■

The following result is due to Berry as reported in [Berry et al., 1986].

**Theorem 2.6.3 (Sequentiality of  $\langle \lambda_V^\rightarrow(\mathbf{A}), \rightarrow \rangle$ ).** *For any program  $M$  which has no  $\rightarrow$ -normal form, exactly one of the following is valid:*

- (i) *For any program  $N$ , whenever  $M \leq_\Omega N$ , then  $N$  too has no  $\rightarrow$ -normal form.*
- (ii) *There is an occurrence  $\beta$  of  $\Omega$  in  $M$  such that for any program  $N$ ,*

$$M \leq_\Omega N \ \& \ N \text{ has a } \rightarrow\text{-normal form} \quad \implies \quad N / \beta \neq \Omega.$$

**Proof.** Given a program  $M$  which has no  $\rightarrow$ -normal form. Suppose

- (1) there is a program  $N_0$  such that  $M \leq_\Omega N_0$  and  $N$  has a  $\rightarrow$ -normal form.

It suffices to prove that there is an occurrence  $\beta$  of  $\Omega$  in  $M$  such that for any program  $N$ , if  $M \leq_\Omega N$  and  $N$  has a  $\rightarrow$ -normal form, then  $N / \beta \neq \Omega$ .

From (1), applying Lemma 2.5.1, we see that  $\text{BT}(M) \in \text{app}^\rightarrow(\mathbf{A})$ . Further, we have

- (2)  $\text{BT}(M) \leq_\Omega \text{BT}(N_0)$     &     $\text{BT}(N_0)$  has a  $\rightarrow_\delta$ -normal form.

By Lemma 2.5.1(iii), we see that  $\text{BT}(M)$  has no  $\rightarrow_\delta$ -normal form. Hence, by the sequentiality of  $\langle \text{app}^\rightarrow(\mathbf{A}), \rightarrow_\delta \rangle$ , there is an occurrence  $\gamma$  such that  $\text{BT}(M) / \gamma = \Omega$  satisfying

- (3) for any program  $P$ , whenever  $\text{BT}(M) \leq_\Omega P$  and  $P$  has a  $\rightarrow_\delta$ -normal form, then  $P / \gamma \neq \Omega$ .

Now, applying the sequentiality of  $\langle \lambda_{\vec{V}}^-(\mathbf{A}), \rightarrow_{\lambda} \rangle$  to  $M$  at  $\gamma$ , on account of (2), and taking  $P$  to be  $\text{BT}(N_0)$  in (3), we deduce  $\text{BT}(N_0) / \gamma \neq \Omega$ ; note also that  $M \leq_{\Omega} N_0$ . Hence, there is a sequentiality index  $\beta$  such that  $\text{BT}(M) / \beta = \Omega$  and that

- (4) for any program  $N$ , whenever  $M \leq_{\Omega} N$  and  $\text{BT}(N) / \gamma \neq \Omega$ , then  $N / \beta \neq \Omega$ .

Now, for any program  $N$  such that  $M \leq_{\Omega} N$  and  $N$  has a  $\rightarrow$ -normal form, by Lemma 2.5.1(iii),  $\text{BT}(M) \leq_{\Omega} \text{BT}(N)$  and  $\text{BT}(N)$  has a  $\rightarrow_{\delta}$ -normal form. Applying (3), we can deduce  $\text{BT}(N) / \gamma \neq \Omega$ . Hence, applying (4), we get  $N / \beta \neq \Omega$ , and we are done. ■

Another important operational property of the reduction system  $\lambda_{\vec{V}}^-(\mathbf{A})$  is *stability*. Like sequentiality, it will be given an abstract treatment in Section 4.

#### Proposition 2.6.4.

- (i) For any  $\lambda_{\vec{V}}^-(\mathbf{A})$ -terms  $M$  and  $T$ . Whenever  $T \leq_{\Omega} \text{BT}(M)$ , then there is a unique  $\leq_{\Omega}$ -least  $M'$  such that  $M' \leq_{\Omega} M$  and that  $T \leq_{\Omega} \text{BT}(M')$ .  
(ii) Let  $A$  be a finite  $\text{app}^-(\mathbf{A})$ -term such that  $A \rightarrow_{\delta} V$  for some ground value  $V$ . Then there is a  $\leq_{\Omega}$ -least  $A'$  satisfying  $A' \leq_{\Omega} A$  and that  $A' \rightarrow_{\delta} V$ . ■

**Theorem 2.6.5 (Stability).** For any program  $P$  such that  $P \rightarrow V$  for some ground value  $V$ , there is a unique  $\leq_{\Omega}$ -least program  $P'$  satisfying  $P' \leq_{\Omega} P$  and  $P' \rightarrow V$ .

**Proof.** By the Separation Proposition, there is some  $Q$  such that  $P \rightarrow_{\lambda} Q$  and  $\omega(Q) \rightarrow_{\delta} V$ . Since  $\omega(Q)$  is a finite  $\text{app}^-(\mathbf{A})$ -term, applying the preceding proposition, we have a unique  $\leq_{\Omega}$ -least  $A$  such that  $A \leq_{\Omega} \omega(Q)$  and  $A \rightarrow_{\delta} V$ . But  $\omega(Q) \leq_{\Omega} \text{BT}(Q) = \text{BT}(P)$ , and so  $A \leq_{\Omega} \text{BT}(P)$ . Hence, applying Proposition 2.6.4(i), there is a unique  $\leq_{\Omega}$ -least  $P'$  such that  $A \leq_{\Omega} \text{BT}(P')$ . This means that for some  $P''$  such that  $P' \rightarrow_{\lambda} P''$ , we have  $A \leq_{\Omega} \omega(P'')$ . Since  $A \rightarrow_{\delta} V$ , we deduce  $\omega(P'') \rightarrow_{\delta} V$ , by an appeal to Lemma 2.4.6. Hence,  $P' \rightarrow V$ . ■

## 2.7 The programming language PCF

PCF is obtained from the reduction system  $\lambda_{\vec{V}}^-(\mathbf{A})$  by specifying a normal order (or leftmost) reduction strategy;  $\beta$ -redexes are contracted in the call-by-name (as opposed to call-by-value) fashion. The strategy permits only *weak* reduction, *i.e.* no reduction is allowed “under a lambda”. The operational semantics is presented in terms of evaluation contexts (see [Felleisen and Friedman, 1986] or [Ong, 1988, §4.4]). We define raw evaluation contexts as

$$E ::= X^{\sigma} \mid (E \cdot M) \mid (f \cdot E)$$

where  $X^\sigma$  denotes a hole of type  $\sigma$ ,  $M$  and  $f$  are meta-variables ranging over PCF-terms and the set  $\mathbf{A}$  of PCF-constants respectively. Note that the hole  $X$  occurs precisely once in any such context  $E$ . An *evaluation context* is just a well-typed raw evaluation context. The *one-step reduction*  $>$  over closed PCF-terms is defined inductively according to the following rules and axioms:

$(\lambda x^\sigma.M)N > M[N/x^\sigma]$	$\mathbf{Y}^\sigma(M) > M(\mathbf{Y}^\sigma(M))$
$\Omega^\sigma > \Omega^\sigma$	$\frac{M > M'}{E[M] > E[M']}$
$\text{cond}^\beta \mathbf{t} MN > M$	$\text{cond}^\beta \mathbf{f} MN > N$
$\text{succ} n > n + 1$	$\text{pred} n + 1 > n$
$\text{pred} 0 > 0$	$\text{zero?} n + 1 > \mathbf{f}$
$\text{zero?} 0 > \mathbf{t}$ .	

Further, we define

$$\begin{aligned}
 &\gg \stackrel{\text{def}}{=} \text{reflexive, transitive closure of } >, \\
 M \gg_n M' &\stackrel{\text{def}}{=} \exists \{M_0, \dots, M_n\}. M \equiv M_0, M_n \equiv M', \text{ \& } M_i > M_{i+1}, \\
 &\quad \text{for all } 0 \leq i < n, \\
 M \Downarrow_n V &\stackrel{\text{def}}{=} M \gg_n V, \text{ for some } n \geq 0, \\
 M \Downarrow V &\stackrel{\text{def}}{=} M \Downarrow_n V, \text{ for some } n \geq 0.
 \end{aligned}$$

Recall that PCF-programs are just closed terms of ground type. Values are  $\lambda$ -abstractions and constants (less  $\Omega$ ), and are ranged over by the meta-variable  $V$ . Following the function paradigm, to compute a program in PCF is to evaluate it.

We may present the operational semantics of PCF equivalently and directly in terms of a Martin-Löf-style evaluation relation (see *e.g.* [Martin-Löf, 1979]). Formally, we define a relation  $\Downarrow$  between closed terms and values recursively over the following rules. We read  $M \Downarrow V$  as “the closed term  $M$  evaluates to the value  $V$ ”.

$V \Downarrow V$	$\frac{M \Downarrow n}{\text{succ}M \Downarrow n+1}$	$\frac{M \Downarrow n+1}{\text{pred}M \Downarrow n}$
$\frac{M \Downarrow 0}{\text{pred}M \Downarrow 0}$	$\frac{M \Downarrow t \quad P \Downarrow V}{\text{cond}^\beta MPQ \Downarrow V}$	$\frac{M \Downarrow f \quad Q \Downarrow V}{\text{cond}^\beta MPQ \Downarrow V}$
$\frac{M \Downarrow 0}{\text{zero?}M \Downarrow t}$	$\frac{M \Downarrow n+1}{\text{zero?}M \Downarrow f}$	$\frac{P[N/x] \Downarrow V}{(\lambda x.P)N \Downarrow V}$
$\frac{MY^\sigma(M) \Downarrow V}{Y^\sigma(M) \Downarrow V}$	$\frac{M \Downarrow V \quad VN \Downarrow V'}{MN \Downarrow V'}$	

Of course, the two notions of  $\Downarrow$  coincide.

**Remark** Plotkin's definition of the programming language PCF differs from ours syntactically in a number of ways.

- For each type  $\sigma$ , our language has a distinguished constant  $\Omega^\sigma$  representing the canonical divergent term of type  $\sigma$ , whereas Plotkin's language does not have them. There are plenty of divergent terms because of the fixed-point operator, *e.g.*  $Y^\sigma(\lambda x^\sigma.x)$ . The presence of undefined terms  $\Omega$  facilitates the formulation of operational properties like sequentiality and stability.
- General recursion is introduced as a unary constructor  $Y^\sigma(-)$ , whereas Plotkin defines fixed-point constants. There is no loss of generality in our definition since the term  $\lambda x^\sigma.Y^\sigma(x^\sigma)$  recovers the effect of a fixed-point constant.
- In our language  $\text{pred}0$  evaluates to 0; in Plotkin's language it does not. Our definition ensures that the only closed ground  $>$ -normal forms are the numerals and the booleans.

### 3 Operational and denotational semantics of PCF

This section concerns the denotational semantics of PCF and the “standard” results of Plotkin, Milner and others on the full abstraction problem for PCF. We begin by examining an operational property of PCF which is sometimes called Context Lemma. The lemma says that observational equivalence between terms is determined by applicative behaviour alone. We then examine what it means for a structure to be a continuous, order-extensional model of PCF. The rest of the section is about the Scott-continuous function space model and its relationship with the operational semantics of PCF. We prove Plotkin's adequacy result and study some structural properties of the continuous, order-extensional, inequationally fully abstract model of PCF which is shown to be unique (up to isomorphism) by Milner. Finally, we prove that the standard Scott-continuous function space model



is fully abstract for PCF extended with a “parallel-or” (or equivalently a “parallel-conditional”) construct.

### 3.1 Context Lemma and observational extensionality

Recall that two program fragments  $M$  and  $N$  are observationally inequivalent just in case there is a program context  $C[X]$  which distinguishes  $M$  from  $N$ ; that is to say, the program  $C[M]$  converges to a ground value,  $V$  say, whereas  $C[N]$  does not (*i.e.*  $C[N]$  either converges to a value distinct from  $V$ , or the computation does not converge), or *vice versa*. More precisely, we say that  $M$  and  $N$  are *observationally equivalent*, written  $M \approx N$ , if for any program context  $C[X]$  such that both  $C[M]$  and  $C[N]$  are programs, and for any value  $V$ ,  $C[M] \Downarrow V$  if and only if  $C[N] \Downarrow V$ . We write  $M \sqsubseteq N$  to mean:

for any program context  $C[X]$  such that both  $C[M]$  and  $C[N]$  are programs, and for any value  $V$ , whenever  $C[M] \Downarrow V$  then  $C[N] \Downarrow V$ .

Of course, in both definitions, we assume that  $M$  and  $N$  are of the same type, and it makes sense to consider only those program contexts  $C[X]$  whose “holes”  $X$  are of the same type as  $M$  and  $N$ .

The Context Lemma states that if  $M$  and  $N$  are observationally inequivalent, then there is a particular kind of program context called *applicative context* which sets them apart. An applicative context is a context which has precisely one hole, and the hole occurs at the leftmost “function” (as opposed to “argument”) position. More precisely, an applicative context is a context of the shape  $C[X] \equiv XM_1 \cdots M_n$  where the  $M_i$ ’s are just PCF-terms.

**Theorem 3.1.1 (Context Lemma).** *Let  $M, N$  be closed PCF-terms of the same type  $\sigma$  which we write as  $(\sigma_1, \dots, \sigma_n, \beta)$ . Whenever  $M$  and  $N$  satisfy the following condition, call it (C):*

*For any closed PCF-terms  $P_1 : \sigma_1, \dots, P_n : \sigma_n$ , and for any value  $V$ , whenever  $MP_1 \cdots P_n \Downarrow V$ , then also  $NP_1 \cdots P_n \Downarrow V$ ;*

*then  $M \sqsubseteq N$ .*

**Proof.** For  $n$  ranging over  $\omega$ , let  $\mathbf{H}(n)$  be the following statement:

*For any closed PCF-terms  $M, N$  of the same type satisfying condition (C), and for any type-compatible program context  $C[X]$ , for any ground value  $V$  whenever  $C[M] \Downarrow_n V$ , then  $C[N] \Downarrow V$ .*

We show by induction on  $n$  that  $\mathbf{H}(n)$  is valid for all  $n \in \omega$ . The base case  $\mathbf{H}(0)$  is trivially true, since the context  $C[X]$  in question must either just be  $X$ , in which case  $M$  is necessarily a value  $V$  say, and so, by condition (C),  $C[N] \equiv N \Downarrow V$ ; or  $C[X]$  is a ground value, in which case the assertion is vacuously true.

We establish the inductive case  $\mathbf{H}(n+1)$  by a syntactic case analysis. It is easy to see that any PCF-program context  $C[X]$  has precisely one of the following syntactic shapes: for  $m \geq 0$ ,

- (1)  $C[X] \equiv \xi C_1[X] \cdots C_m[X]$  where  $\xi$  is either  $\Omega$  or an element of the set  $\mathbf{A}$  of arithmetic constants,
- (2)  $C[X] \equiv (\lambda x^\sigma. C_0[X]) C_1[X] \cdots C_m[X]$ ,
- (3)  $C[X] \equiv \mathbf{Y}^\sigma(C_0[X]) C_1[X] \cdots C_m[X]$ ,
- (4)  $C[X] \equiv X C_1[X] \cdots C_m[X]$ .

(1) The case of  $\xi \equiv \Omega$  is vacuous. We consider the case of  $\xi \equiv \text{cond}^c$  just for illustration. Suppose  $\text{cond}^c C_1[M] C_2[M] C_3[M] \Downarrow_{n+1} V$ . Then, we must have  $C_1[M] \Downarrow_{l_1} t$  or  $f$ . Suppose the former is the case; we then have  $C_2[M] \Downarrow_{l_2} V$ . Note that both  $l_1$  and  $l_2 \leq n$ . Hence, by the induction hypothesis, we have  $C_1[N] \Downarrow t$  and  $C_2[N] \Downarrow V$ ; and so  $\text{cond}^c C_1[N] C_2[N] C_3[N] \Downarrow V$ . The argument is entirely similar for the case of  $C_1[M] \Downarrow_{l_1} f$ .

(2) Note that we must have  $m \geq 1$ . Suppose

$$C[M] \equiv (\lambda x^\sigma. C_0[M]) C_1[M] \cdots C_m[M] \Downarrow_{n+1} V.$$

Then  $C[M] > C_0[M][C_1[M]/x^\sigma] C_2[M] \cdots C_m[M] \Downarrow_n V$ .

Let  $D[X] \equiv C_0[X][C_1[X]/x^\sigma] C_2[X] \cdots C_m[X]$ . Since  $M$  and  $N$  are closed terms,  $C[M] > D[M] \Downarrow_n V$ . By  $\mathbf{H}(n)$ , we infer that  $D[N] \Downarrow V$ ; and so  $C[N] \Downarrow V$ .

(3) Suppose  $C[M] \equiv \mathbf{Y}^\sigma(C_0[M]) C_1[M] \cdots C_m[M] \Downarrow_{n+1} V$ . Define a new context

$$D[X] \equiv C_0[X] \mathbf{Y}^\sigma(C_0[X]) C_1[X] \cdots C_m[X].$$

We can infer that  $C[M] > D[M] \Downarrow_n V$ . By  $\mathbf{H}(n)$ , we have  $D[N] \Downarrow V$ . Since  $C[N] > D[N]$ , we conclude that  $C[N] \Downarrow V$ .

(4) Suppose  $C[M] \equiv M C_1[M] \cdots C_m[M] \Downarrow_{n+1} V$ . Observe that  $M$  must have one of the following syntactic shapes:

- (a)  $\xi \vec{A}$  where  $\xi$  is either  $\Omega$  or an arithmetic constant,
- (b)  $(\lambda x. A) \vec{B}$ ,
- (c)  $\mathbf{Y}(A) \vec{B}$ .

Let  $D[X] \equiv M C_1[X] \cdots C_m[X]$ ; we then have  $C[M] \equiv D[M] \Downarrow_{n+1} V$ . Note that if  $M$  has shape (a) (or (b), or (c) respectively), then  $D[X]$  is a context of type (1) (or (2), or (3) respectively). Applying  $\mathbf{H}(n+1)$  to contexts of type (1), (2) or (3) as the case may be, we infer that  $D[N] \Downarrow V$ . But  $D[N] \equiv M C_1[N] \cdots C_m[N]$ ; by condition (C), so we have  $N C_1[N] \cdots C_m[N] \equiv C[N] \Downarrow V$ . ■

Meyer refers to the Context Lemma as the Operational Extensionality Theorem [Meyer and Cosmadakis, 1988]. The gist of the Context Lemma

is this: terms of the language PCF are determined by their applicative behaviour. Hence, it is quite appropriate to call PCF an applicative or functional programming language. The proof we have presented is due to Berry [Berry, 1979]. Milner [Milner, 1977] has a similar result for a family of simply typed PCF-like languages expressed in the formalism of combinatory logic. An interesting line of research is to establish a general Context Lemma. This is the problem of finding conditions under which the Context Lemma is valid for a general class of programming languages. An example of this kind of result has been obtained by Jim and Meyer [Jim and Meyer, 1991]. They showed that the Context Lemma is valid in any simply typed lambda calculus with a “PCF-like” rewriting semantics.

**Remark** Observational equivalence captures a very intuitive notion of program equality which is based on the concept of observable behaviour universally quantified over program contexts. There is another intuitively appealing notion of program equivalence for both typed and untyped functional languages called *applicative bisimulation* which is inspired by Milner’s idea of bisimulation in process algebra (see *e.g.* [Milner, 1989] and [Park, 1980]). Applicative bisimulation was introduced by Abramsky [Abramsky, 1990] in the study of the lazy lambda calculus. There is a close relationship between applicative bisimulation and observational equivalence. For example, the following are equivalent in the lazy lambda calculus (see [Abramsky and Ong, 1993]):

- (1) applicative bisimilarity implies observational equivalence (the reverse implication is trivially valid),
- (2) the Context Lemma is valid,
- (3) applicative bisimulation is a congruence.

### 3.2 Denotational models

What is a denotational model of the programming language PCF? We first give an informal description. Since PCF is a typed functional language, an interpretation of the language must first provide a system of “domains” which gives an interpretation of the simple types, one domain for each type. The denotation of a PCF-term is then given by a type-respecting valuation function which takes a PCF-term and an environment giving denotations of the variables occurring free in the term to produce a denotation of the term as an element of the appropriate domain. Following the style of denotational semantics, the valuation function should be defined compositionally.

As is well-known, models of the simply typed  $\lambda$ -calculus are just<sup>10</sup> cartesian closed categories. The interpretation is standard: given a cartesian closed category, types of the calculus are interpreted as objects, and terms

---

<sup>10</sup>To be exact, there is an equivalence between the category of simply-typed  $\lambda$ -calculi with surjective pairing, and the category of cartesian closed categories; see *e.g.* [Lambek and Scott, 1986].

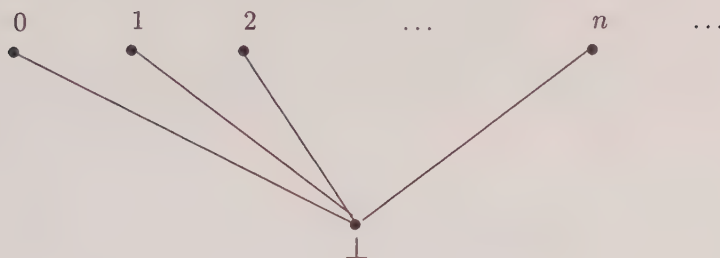


Fig. 1. The flat CPO of natural numbers.

as morphisms. Since PCF is a simply typed  $\lambda$ -calculus, we look for a cartesian closed category of domains. An important feature of PCF is general recursion in the form of a fixed-point operator at each function type of the shape  $\sigma \Rightarrow \sigma$ . As is standard, we appeal to Knaster-Tarski's Fixed-point Theorem [Tarski, 1955] to interpret the fixed-point operator of a continuous function from a complete partial order (CPO) to itself. It is just the least upper bound of an  $\omega$ -increasing sequence of successive iterates of the function. Therefore, in order to carry this programme through, we require the domains to be CPOs (formal definition to follow). Also, we insist that the valuation function be continuous.

A *collection of value domains* for PCF comprises

- a family  $\{D^\sigma\}$  of CPOs, one for each type  $\sigma$ ,
- for each pair  $\sigma$  and  $\tau$  of types, an application operation  $\cdot : D^{\sigma \Rightarrow \tau} \times D^\sigma \rightarrow D^\tau$ . For  $d \in D^{\sigma \Rightarrow \tau}$  and  $e \in D^\sigma$ , we often write  $d \cdot e$  simply as  $de$ .

Recall that a CPO is a partial order such that every directed subset has a least upper bound (lub). We write the least element of  $D^\sigma$  as  $\perp^\sigma$ , or simply  $\perp$ .

An *interpretation* of the language  $\lambda_{\vec{Y}}^-(\mathbf{A})$  is a pair  $\langle \{D^\sigma\}, \mathcal{A} \rangle$  where  $\{D^\sigma\}$  is a collection of value domains for PCF and the map

$$\mathcal{A} : \mathbf{A} \longrightarrow \bigcup_{\sigma} D^\sigma$$

is type-respecting, *i.e.* if  $c \in \mathbf{A}$  is a constant of type  $\sigma$ , then  $\mathcal{A}(c)$  is an element of  $D^\sigma$ . Such an interpretation  $\mathcal{A}$  (of  $\mathbf{A}$ ) is said to be *standard* if  $D^\iota$  is the flat CPO of natural numbers, ordered as in Figure 1 and  $D^o$  is the flat CPO of booleans ordered as in Figure 2; together with the following interpretation of constants: for  $d, e \in D^\beta$ ,  $n \in D^\iota$ ,  $b \in D^o$ ,



Fig. 2. The flat CPO of boolean values.

$$\mathcal{A}(t) = t$$

$$\mathcal{A}(f) = f$$

$$\mathcal{A}(\text{cond}^\beta)bde = \begin{cases} d & \text{if } b = t, \\ e & \text{if } b = f \\ \perp & \text{if } b = \perp \end{cases}$$

$$\mathcal{A}(\text{zero?})n = \begin{cases} t & \text{if } n = 0 \\ f & \text{if } n > 0 \\ \perp & \text{if } n = \perp \end{cases}$$

$$\mathcal{A}(\text{succ})n = \begin{cases} n + 1 & \text{if } n \geq 0 \\ \perp & \text{if } n = \perp \end{cases}$$

$$\mathcal{A}(\text{pred})n = \begin{cases} n - 1 & \text{if } n \geq 1 \\ 0 & \text{if } n = 0 \\ \perp & \text{if } n = \perp \end{cases}$$

$$\mathcal{A}(n) = n \quad (n \geq 0).$$

We need the set  $\text{Env}$  of *environments*. An environment is a type-respecting function from the set of variables to  $\cup_{\sigma} D^{\sigma}$ . Environments are ranged over by the meta-variable  $\rho$ . As usual, the update operation is defined as follows: for  $d \in D^{\sigma}$ ,

$$\rho[x^{\sigma} \mapsto d](y) \stackrel{\text{def}}{=} \begin{cases} d & \text{if } y = x^{\sigma}, \\ \rho(x) & \text{otherwise.} \end{cases}$$

The *undefined environment*  $\perp$  sends every variable  $x^{\sigma}$  to  $\perp^{\sigma}$ .

**Order-extensional, continuous model of PCF** Given an interpretation  $\langle \{D^{\sigma}\}, \mathcal{A} \rangle$ , a *continuous, least fixed-point model* (or simply *continuous model*) of PCF is a type-respecting semantic function



$$\mathcal{A}[-](-) : \lambda_{\mathbf{V}}^{\rightarrow}(\mathbf{A}) \longrightarrow (\text{Env} \longrightarrow \bigcup_{\sigma} D^{\sigma})$$

satisfying three sets of criteria under the following headings:

- compositional, least fixed-point semantics,
- structural constraints,
- operational soundness.

Further, the model is said to be *order-extensional* if for any arrow type  $\sigma \Rightarrow \tau$ , elements of the value domain  $D^{\sigma \Rightarrow \tau}$  are ordered extensionally; more formally, for any  $f, g \in D^{\sigma \Rightarrow \tau}$ ,  $f \sqsubseteq g$  if and only if  $f \cdot d \sqsubseteq g \cdot d$  for all  $d \in D^{\sigma}$ . The model is *extensional* if for any  $f, g \in D^{\sigma \Rightarrow \tau}$ ,  $f = g$  if and only if  $f \cdot d = g \cdot d$  for all  $d \in D^{\sigma}$ . The condition of extensionality is just another way of saying that for each function type  $\sigma \Rightarrow \tau$ , the value domain  $D^{\sigma \Rightarrow \tau}$  is a collection of functions from  $D^{\sigma}$  to  $D^{\tau}$ . It is easy to see that a model is extensional if it is order-extensional.

**Compositional, least fixed-point semantics** For  $M$  and  $N$  ranging over PCF-terms, and  $d \in D^{\sigma}$ ,

$$\begin{aligned} \mathcal{A}[x^{\sigma}](\rho) &= \rho(x^{\sigma}), \\ \mathcal{A}[\Omega^{\sigma}](\rho) &= \perp^{\sigma}, \\ \mathcal{A}[c^{\sigma}](\rho) &= \mathcal{A}(c^{\sigma}), \\ \mathcal{A}[MN](\rho) &= \mathcal{A}[M](\rho) \cdot \mathcal{A}[N](\rho), \\ \mathcal{A}[\lambda x^{\sigma}.M](\rho) \cdot d &= \mathcal{A}[M](\rho[x^{\sigma} \mapsto d]), \\ \mathcal{A}[\mathbf{Y}^{\sigma}(M)](\rho) &= \text{fix}_{\sigma}(\mathcal{A}[M](\rho)); \end{aligned}$$

where  $\text{fix}_{\sigma}(f)$  is a shorthand for  $\bigsqcup \{ f^n(\perp^{\sigma}) : n \in \omega \}$ , for any  $f \in D^{\sigma \Rightarrow \sigma}$ . As usual, we write  $f^0(\perp)$  to mean  $\perp$  and  $f^n(\perp)$  to mean  $\underbrace{f(\cdots (f(\perp)) \cdots)}_n$ .

Our interpretation of the fixed-point operator  $\mathbf{Y}^{\sigma}(-)$  relies on Knaster-Tarski's Fixed-point Theorem (see *e.g.* the account in [Winskel, 1993]). In order to apply the theorem, we require the denotation of any closed PCF-term  $M$  of function type  $\sigma \Rightarrow \sigma$  (which we already know is an element of  $D^{\sigma \Rightarrow \sigma}$ ) to be a continuous function from  $D^{\sigma}$  to  $D^{\sigma}$ . This forces us to place an additional constraint on the collection of all value domains: for each function type  $\sigma \Rightarrow \tau$ , the value domain  $D^{\sigma \Rightarrow \tau}$  should be a subCPO of the CPO  $[D^{\sigma} \Rightarrow D^{\tau}]$  of continuous functions from  $D^{\sigma}$  to  $D^{\tau}$ .

**Structural constraints** The valuation function should be a well-behaved function of the environment. It should interact sensibly with the operations of term-substitution and contextual-substitution: for any (well-typed) terms  $M$  and  $N$ , and any  $\rho, \rho' \in \text{Env}$ ,

- (Env) if  $\rho(x^\sigma) = \rho'(x^\sigma)$  for all  $x^\sigma \in \text{FV}(M)$ ,  
 then  $\mathcal{A}[M](\rho) = \mathcal{A}[M](\rho')$ ;
- (Subst)  $\mathcal{A}[M[N/x^\sigma]](\rho) = \mathcal{A}[M](\rho[x^\sigma \mapsto \mathcal{A}[N](\rho)])$ ;
- (Cont) if  $\mathcal{A}[M](\rho) = \mathcal{A}[N](\rho)$  for all  $\rho$ ,  
 then for any type-compatible context  $C[X]$ ,  
 we have  $\mathcal{A}[C[M]](\rho) = \mathcal{A}[C[N]](\rho)$  for all  $\rho$ .

The first structural constraint, (Env), is entirely natural: the valuation of a term should clearly be invariant over environments which agree on all variables occurring free in the term. For this reason, so long as the term  $M$  is closed (*i.e.* has no free variables), there is no harm in writing the denotation of  $M$  as  $\mathcal{A}[M](\perp)$ , where  $\perp$  is the environment which maps every variable to  $\perp$ ; in fact, we shall sometimes be even sloppier and write it as  $\mathcal{A}[M]$ .

The second structural constraint, (Subst), together with the preceding set of least fixed-point semantics constraints, validate  $\beta$ -equivalence; that is to say, for any PCF-terms  $M : \tau$  and  $N : \sigma$ , and for any environment  $\rho$ ,

$$\mathcal{A}[(\lambda x^\sigma.M)N](\rho) = \mathcal{A}[M[N/x^\sigma]](\rho).$$

This is an assertion which the reader may wish to check as an easy exercise. Finally, the third structural constraint, (Cont), asserts that the valuation of PCF-terms is a “context-free” operation.

**Operational soundness** If  $M \rightarrow N$ , then  $\mathcal{A}[M](\rho) = \mathcal{A}[N](\rho)$  for all  $\rho$ .

Assuming that the interpretation of the ground type is standard, this constraint ensures that whenever a program converges to a value, its denotation coincides with it.

### 3.3 Adequacy

What does it mean to say that two programs or program fragments are equivalent? As we have seen, there is a purely “symbol-pushing”, operational notion of program equivalence called *observational equivalence* which programmers readily accept as fundamental: two program fragments are equivalent if they can always be *interchanged* without affecting the visible or observable outcome of the computation. In contrast, according to denotational semantics, the meaning of a program or a term is what the semantic function denotes; it is an element in the denotational model. Hence, two programs are equal in the denotational sense if they have the same denotation in the model.

A major theme of this chapter is the correspondence between observational (or operational) and denotational views of program equivalence. A

denotational semantics is said to be *adequate* for a programming language if denotational equality implies observational equivalence. This implication is often the easier of the two directions to prove. If implication in the other direction holds as well, then the denotational semantics is said to be (equationally) *fully abstract* for the language. Operational behaviour of a programming language may be related to the denotational model in an even weaker sense: we say that the denotational semantics is *weakly adequate* if for any program  $M$  and for any value  $V$ ,  $\mathcal{A}[\![M]\!](\perp) = \mathcal{A}[\![V]\!](\perp)$  if and only if  $M \Downarrow V$ .

Weak adequacy expresses such a loose correspondence between denotational and operational behaviour of programs that if *even* it does not hold, then the denotational semantics in question is probably not worth having at all. Weak adequacy is thus *sine qua non* in evaluating how well-fitted a denotational semantics is to a programming language. If the valuation function is compositional, or more precisely, if the structural constraint, **(Cont)**, is satisfied, then weak adequacy implies adequacy. It is easy to see why this is so. Take any PCF-terms  $M$  and  $N$ , and suppose  $\mathcal{A}[\![M]\!](\rho) = \mathcal{A}[\![N]\!](\rho)$  for any environment  $\rho$ . Given any program context  $C[X]$  such that both  $C[M]$  and  $C[N]$  are programs, suppose  $C[M] \Downarrow V$  for some value  $V$ . By weak adequacy,  $\mathcal{A}[\![C[M]]\!](\perp) = \mathcal{A}[\![V]\!](\perp)$ . By the structural constraint, **(Cont)**, we have  $\mathcal{A}[\![C[M]]\!](\perp) = \mathcal{A}[\![C[N]]\!](\perp)$ . Hence, we deduce  $\mathcal{A}[\![C[N]]\!](\perp) = \mathcal{A}[\![V]\!](\perp)$  for any program context  $C[X]$ . By weak adequacy again, we have  $C[N] \Downarrow V$ . In exactly the same way, we can show that  $C[N] \Downarrow V$  implies  $C[M] \Downarrow V$ . Hence, we are done.

**Theorem 3.3.1 (Weak Adequacy).** *Let  $\mathcal{A}[\!-\!](-)$  be a continuous model of PCF following the standard interpretation. For any program  $M$  and any ground value  $V$ , whenever  $\mathcal{A}[\![M]\!](\perp) = \mathcal{A}[\![V]\!](\perp)$  then  $M \Downarrow V$ . ■*

The proof strategy is based on a so-called *computability argument* in [Plotkin, 1977]. It illustrates a salient feature of inductive reasoning in typed  $\lambda$ -calculus: it is often necessary to establish a property *uniformly* (for example, one that is applicable to all terms as opposed to just the closed terms; or to all types as opposed to just ground types) for all terms when we may only be interested in its validity for a proper subcollection of terms.

**A computability argument** Let  $V$  range over ground values. In the following, for closed PCF-terms  $M$ , we shall write the denotation  $\mathcal{A}[\![M]\!](\perp)$  simply as  $\mathcal{A}[\![M]\!]$ , omitting the environment argument  $\perp$ . Our task now is to define a predicate called *computability* over *all* PCF-terms. At ground type, a closed term (= a program) is computable if and only if it satisfies the assertion in the above Weak Adequacy Theorem. The notion of computability is then extended to terms of higher type as well as to terms which may have free variables. This uniform extension of the definition of computability to all PCF-terms is crucial to the computability argument.

- If  $M : \beta$  is closed, then  $M$  is computable if  $\mathcal{A}[M] = \mathcal{A}[V]$  implies  $M \Downarrow V$ .
- If  $M : \sigma \Rightarrow \tau$  is closed, then  $M$  is computable just in case  $MN$  is computable for every closed computable term  $N : \sigma$ .
- If  $M : \sigma$  has free variables  $\{x_1^{\sigma_1}, \dots, x_n^{\sigma_n}\}$ , then  $M$  is computable if for every substitution  $\theta$  mapping each  $x_i^{\sigma_i}$  to a closed computable term  $N_i$ , the term

$$M_\theta \equiv M[N_1/x_1^{\sigma_1}, \dots, N_n/x_n^{\sigma_n}]$$

is computable.

Note that we regard a substitution as a type-respecting map from variables to terms. Combining the above, we observe that a term  $M : (\sigma_1, \dots, \sigma_n, \beta)$  is *computable* if and only if

- for any substitution  $\theta$  mapping free variables of  $M$  to closed computable terms, and
- for any closed computable terms  $N_1 : \sigma_1, \dots, N_n : \sigma_n$ ,

the term  $M_\theta N_1 \dots N_n : \beta$  is computable.

**Proposition 3.3.2.** *For any PCF-term  $M$ ,  $M$  is computable.*

**Proof.** We prove by induction on the structure of  $M : \sigma$ . Recall that  $\sigma = (\sigma_1, \dots, \sigma_n, \beta)$ . A substitution  $\theta$  is said to be *closed* and *computable* for  $M$  if  $\theta$  maps the free variables of  $M$  to closed, computable terms.

CASE  $M \equiv \Omega^\sigma$ : This is vacuously true since  $\mathcal{A}[\Omega^\sigma \vec{N}] = \perp$  for any sequence of closed terms  $N_i : \sigma_i$ .

CASE  $M \equiv x^\sigma$ : For any closed, computable substitution  $\theta$  for  $x^\sigma$  and for any sequence of closed computable terms  $N_i : \sigma_i$ , suppose  $\mathcal{A}[x_\theta \vec{N}] = \mathcal{A}[V]$ . Since  $x_\theta$  is computable by supposition, we have  $x_\theta \vec{N} \Downarrow V$  as desired.

CASE  $M \equiv c$  for  $c \in \mathbf{A}$ : We just consider the case of  $\text{cond}^\beta$  for illustration. For any closed, computable terms  $N_2, N_3$  of type  $\beta$  and  $N_1$  of type  $\sigma$ , suppose  $\mathcal{A}[\text{cond}^\beta N_1 N_2 N_3] = \mathcal{A}[V]$ . By definition of the standard interpretation, w.l.o.g. say  $\mathcal{A}[N_1] = \mathbf{t}$ , and so  $\mathcal{A}[N_2] = \mathcal{A}[V]$ . Since  $N_1$  and  $N_2$  are closed, computable terms, we have  $N_1 \Downarrow \mathbf{t}$  and  $N_2 \Downarrow V$ . Hence,  $\text{cond}^\beta N_1 N_2 N_3 \Downarrow V$ .

CASE  $M \equiv \lambda x^{\sigma_1}.P$ : For any closed, computable substitution  $\theta$  for  $M$ , and for any sequence of closed computable terms  $N_i : \sigma_i$ , suppose

$$\mathcal{A}[(\lambda x^{\sigma_1}.P)_\theta \vec{N}] = \mathcal{A}[V].$$

Note that the following are valid:

- (A)  $\mathcal{A}[(\lambda x^{\sigma_1}.P)_\theta N_1] = \mathcal{A}[(P_\theta[N_1/x^{\sigma_1}])] = \mathcal{A}[(P[N_1/x^{\sigma_1}])_\theta]$ ; and
- (B)  $(P[N_1/x^{\sigma_1}])_\theta = P_{\theta'}$  where  $\theta' \stackrel{\text{def}}{=} \theta[x^{\sigma_1} \mapsto N_1]$ .

Putting (A) and (B) together, we observe that



$$\mathcal{A}[P_{\theta'} N_2 \cdots N_n] = \mathcal{A}[(\lambda x^{\sigma_1}. P)_{\theta} N_1 \cdots N_n] = \mathcal{A}[V].$$

By the induction hypothesis,  $P$  is computable; and so, since the substitution  $\theta'$  is closed and computable for  $P$ , we have  $P_{\theta'} N_2 \cdots N_n \Downarrow V$ . By an appeal to (B), we have  $(\lambda x^{\sigma_1}. P_{\theta}) \vec{N} \Downarrow V$ .

CASE  $M \equiv \mathbf{Y}^{\sigma}(P)$ : This case requires us to introduce syntactic approximants  $\mathbf{Y}_l^{\sigma}(M)$  of the fixed-point operator where  $l$  ranges over  $\omega$ . The language PCF is extended syntactically by the following rule: for every  $l \in \omega$ ,

$$\frac{M : \sigma \Rightarrow \sigma}{\mathbf{Y}_l^{\sigma}(M) : \sigma}.$$

We extend the rules for the reduction with the following:

$$\begin{aligned} \mathbf{Y}_0^{\sigma}(M) &> \mathbf{Y}_0^{\sigma}(M) \\ \mathbf{Y}_{l+1}^{\sigma}(M) &> M \mathbf{Y}_l^{\sigma}(M). \end{aligned}$$

It is easy to see what the denotation of the approximant should be. We extend the inductive definition by the following rule:

$$\mathcal{A}[\mathbf{Y}_l^{\sigma}(M)] = f^l(\perp),$$

where  $f$  is  $\mathcal{A}[M]$ . We note that

$$\mathcal{A}[\mathbf{Y}_{l+1}^{\sigma}(M)] = \mathcal{A}[M \mathbf{Y}_l^{\sigma}(M)].$$

We need the following claim whose proof we leave as an exercise to the reader: **Claim**

- (i) For any  $M : \sigma$ ,  $\mathcal{A}[\mathbf{Y}^{\sigma}(M)] = \bigsqcup_{l \in \omega} \mathcal{A}[\mathbf{Y}_l^{\sigma}(M)]$ .
- (ii) Whenever  $\mathbf{Y}_l^{\sigma}(P) \vec{N} \Downarrow J : \beta$  for some  $l$ , then  $\mathbf{Y}^{\sigma}(P) \vec{N} \Downarrow J$ .

For any closed, computable substitution  $\theta$  for  $P$ , and for any sequence of closed, computable terms  $N_i : \sigma_i$ , suppose  $\mathcal{A}[\mathbf{Y}^{\sigma}(P_{\theta}) \vec{N}] = \mathcal{A}[V]$ . By Claim(i) and by the compositionality of the valuation function  $\mathcal{A}[-]$ , we have

$$\begin{aligned} \mathcal{A}[V] &= (\bigsqcup_{l \in \omega} \mathcal{A}[\mathbf{Y}_l^{\sigma}(P_{\theta})]) \mathcal{A}[N_1] \cdots \mathcal{A}[N_n] \\ &= \bigsqcup_{l \in \omega} \mathcal{A}[(\mathbf{Y}_l^{\sigma}(P_{\theta})) N_1 \cdots N_n]. \end{aligned}$$

The last step is justified because application is a continuous operation. Now, since  $D^{\beta}$  is a flat cpo,  $\mathcal{A}[(\mathbf{Y}_l^{\sigma}(P_{\theta})) N_1 \cdots N_n] = \mathcal{A}[V]$  for some  $l$ . Since  $\mathbf{Y}_l^{\sigma}(P)$  is computable, we have  $(\mathbf{Y}_l^{\sigma}(P_{\theta})) N_1 \cdots N_n \Downarrow V$ . By Claim(ii), we have the desired result  $(\mathbf{Y}^{\sigma}(P_{\theta})) N_1 \cdots N_n \Downarrow V$ .

CASE  $M \equiv \mathbf{Y}_l^{\sigma}(P)$ , for each  $l \in \omega$ : We prove by induction on  $l$ . The base case of  $l = 0$  is obvious. Take as our induction hypothesis the computability



of  $\mathbf{Y}_l^\sigma(P)$ . For any closed, computable substitution  $\theta$  for  $\mathbf{Y}_{l+1}^\sigma(P)$ , and any sequence of closed, computable terms  $N_i : \sigma_i$ , suppose

$$\mathcal{A}[\mathbf{Y}_{l+1}^\sigma(P_\theta)\vec{N}] = \mathcal{A}[V].$$

Since  $\mathbf{Y}_{l+1}^\sigma(P_\theta) > P_\theta(\mathbf{Y}_l^\sigma(P_\theta))$ , and the semantic function is sound, we have

$$\mathcal{A}[P_\theta(\mathbf{Y}_l^\sigma(P_\theta))\vec{N}] = \mathcal{A}[V].$$

By the main induction hypothesis,  $P$  is computable, and so

$$P_\theta(\mathbf{Y}_l^\sigma(P_\theta))\vec{N} \Downarrow V.$$

Hence,  $\mathbf{Y}_{l+1}^\sigma(P_\theta)\vec{N} \Downarrow V$ .

CASE  $M \equiv PQ$ : Consider the possible syntactic shapes of  $M$ :

- $\Omega^v A\vec{B}$ ,
- $x^v A\vec{B}$ ,
- $c^v A\vec{B}$  for  $c^v \in \mathbf{A}$ ,
- $(\lambda x^v. P)A\vec{B}$ ,
- $(\mathbf{Y}_n^v(P))A\vec{B}$ ,
- $(\mathbf{Y}^v(P))A\vec{B}$ .

Consider the last case for illustration. For any closed, computable substitution  $\theta$  for  $(\mathbf{Y}^v(P))A\vec{B}$ , and for any sequence of closed, computable terms  $N_i : \sigma_i$ , suppose  $\mathcal{A}[(\mathbf{Y}^v(P))A\vec{B}_\theta\vec{N}] = \mathcal{A}[V]$ . By the induction hypothesis,  $A_\theta$  and  $\vec{B}_\theta$  are closed and computable, and so, since  $\mathbf{Y}^v(P)$  is computable as established in the previous case,  $((\mathbf{Y}^v(P))A\vec{B}_\theta)\vec{N} \Downarrow V$ . The rest are entirely similar. ■

The computability argument central to the proof belongs to a family of inductive arguments over typed structures which trace their intellectual origin back to an important paper of Tait [Tait, 1967]. Building on Tait's idea, Girard developed a powerful proof technique called candidates of reducibility. To prove the strong normalization of *System F* [Girard, 1972] (see the lucid account in Girard's book [Girard *et al.*, 1989]). Another variation of the same theme called logical relation was independently developed by Plotkin [Plotkin, 1972]; see also [Statman, 1985]. Logical relations are a very useful tool for proving properties of typed lambda-calculi.

Denotational semantics prescribes meanings to syntactic entities *compositionally*. This means that the meaning of a composite term is defined in terms of the meanings of the respective components. If the mathematical properties of the underlying denotational model are well understood, then the denotational approach lends itself to an "algebraic" mode of reasoning about global properties of programs. However, for the semantic

foundations provided by the denotational model to be *relevant* to the programming language in question, the denotational model has to fit well with the operational behaviour of programs. Adequacy establishes just such a “goodness-of-fit” criterion which is of fundamental importance. Plotkin’s adequacy result for PCF has been extended to richer computational settings. For example, in [Plotkin, 1985], it is shown that adequacy holds for a meta-language for denotational semantics which has an elaborate type system including recursive types. An interesting line of research is to prove adequacy for as general a computational setting as possible. A related problem is the investigation of proof techniques for adequacy. For example, Pitts in [Pitts, 1994] applies a general and powerful machinery mixing inductive and co-inductive reasoning to investigate adequacy proofs with respect to domains defined by recursive domain equations.

### 3.4 Order-extensional, continuous, fully abstract model

A denotational semantics of a programming language is said to be (equationally) *fully abstract* just in case the denotational notion of program equality coincides with observational equivalence. A main theme of the rest of this section is the structural properties of order-extensional, continuous models of PCF. We shall prove that any order-extensional, continuous model of PCF following the standard interpretation is necessarily a system of Scott domains. This lays the foundation for a major result of Plotkin and Milner:

*An order-extensional, continuous model of PCF is fully abstract if and only if every finite element of the system of values domains is PCF-definable.* ■

**Technical preliminaries** A *complete partial order* (or CPO, for short) is a structure  $\langle D, \sqsubseteq \rangle$  such that  $D$  is a set partially ordered by  $\sqsubseteq$  which has a least element  $\perp$ , and that every directed subset of  $D$  has a least upper bound (lub) in  $D$ . Recall that a subset  $X$  of a poset  $\langle D, \sqsubseteq \rangle$  is *directed* just in case any pair of elements  $x, y$  of  $X$  has an upper bound in  $X$ . An element  $d$  of a CPO  $D$  is *compact* (or finite, or isolated) if for any directed subset  $X$  of  $D$ , whenever  $d \sqsubseteq \bigsqcup X$ , then there is some  $x \in X$  such that  $d \sqsubseteq x$ . A CPO  $D$  is *algebraic* if for every element  $d \in D$ , the subset  $\{x \sqsubseteq d : x \text{ is compact}\}$  of  $D$  is directed, and its lub is  $d$ . Further,  $D$  is  $\omega$ -*algebraic* if  $D$  has denumerably many compact elements. A subset  $X$  of the CPO  $D$  is said to be *consistent* (or compatible) if it has an upper bound in  $D$ , and we write  $\uparrow X$ , or  $x \uparrow y$  if  $X$  is the set  $\{x, y\}$ . For any CPO  $D$ , the following properties are equivalent:

- (1) *consistent completeness*: every consistent subset of  $D$  has an lub in  $D$ ,
- (2) every compatible pair of elements of  $D$  has an lub in  $D$ ,
- (3) every non-empty subset has a greatest lower bound (glb) in  $D$ .

A CPO is said to be *consistently complete* if any one of the above conditions is satisfied. We shall call any consistently complete,  $\omega$ -algebraic CPO a *Scott domain*. A key property of the category of Scott domains is that it is closed under function space construction. In fact, it is a cartesian closed category. A good reference is [Plotkin, 1981a]; see also [Davey and Priestley, 1990].

An *inverse system of projections* of a CPO  $D$  is a countable collection  $\langle \psi_n \rangle_{n \in \omega}$  of continuous functions from  $D$  to  $D$  satisfying the following conditions: for each  $n \in \omega$ ,

- *projection*:  $\psi_n \circ \psi_n = \psi_n$ ,
- *finiteness*: the image  $\psi_n(D)$  is finite (and hence a finite CPO),
- *approximation*:  $\psi_n \sqsubseteq \psi_{n+1}$ ,
- $\omega$ -*limit*:  $\text{id}_D = \bigsqcup_{n \in \omega} \psi_n$ .

We leave it as an instructive exercise for the reader to show that if a CPO  $D$  has an inverse system of projections, then it is  $\omega$ -algebraic, and that  $\{\psi_n d : n \geq 0, d \in D\}$  is the set of compact elements of  $D$ . In fact, given any CPO  $D$ , the following conditions are equivalent:

- (1)  $D$  has an inverse system of projections.
- (2)  $D$  is the colimit of a directed diagram of finite posets over the category of CPOs and embedding-projection pairs.
- (3)  $D$  is  $\omega$ -algebraic, and satisfies
  - \* *Condition (M)*: for any finite subset  $X$  of compact elements of  $D$ , the set  $\text{MUB}(X)$  of minimal upper bounds of  $X$  is finite, and, moreover, *complete* in the sense that for any upper bound  $u$  of  $X$ , there is some  $x \in \text{MUB}(X)$  such that  $x \sqsubseteq u$ .
  - \* Further, the set  $\mathcal{U}^*(X)$  is finite, where

$$\begin{aligned} \mathcal{U}^0(X) &\stackrel{\text{def}}{=} X, \\ \mathcal{U}^{n+1}(X) &\stackrel{\text{def}}{=} \bigcup \{ \text{MUB}(U) : U \subseteq \mathcal{U}^n(X) \}, \\ \mathcal{U}^*(X) &\stackrel{\text{def}}{=} \bigcup_{k \in \omega} \mathcal{U}^k(X). \end{aligned}$$

A CPO  $D$  is said to be *strongly algebraic* if any one of the above conditions is satisfied. A strongly algebraic CPO is also known as an SFP domain [Plotkin, 1976]. The name is an acronym for “Sequence of Finite Posets” which arises from one of the main descriptions of an SFP domain as the bilimit (i.e. a direct limit of embeddings, or equivalently, by the “limit-colimit coincidence” due to Scott, an inverse limit of projections) of a sequence of finite posets. For this reason, an SFP domain is also called a *bifinite* CPO. The following references are recommended: [Plotkin, 1981a], [Gunter and Scott, 1990] or [Gunter, 1992, Ch. 10]. We leave the proof of the following lemma as an exercise to the reader:

**Lemma 3.4.1.** *An SFP domain in which any two elements have a greatest lower bound is a Scott domain.* ■

Suppose a collection  $\{D^\sigma\}$  of CPOs as value domains for PCF constitutes a continuous, order-extensional model of PCF. What properties can we infer about the structure of the system of CPOs  $\{D^\sigma\}$ ? A result in the following states that such value domains are Scott domains. We first state an important necessary and sufficient condition for an orderextensional, continuous model for PCF to be fully abstract.

**Theorem 3.4.2 (Definability).** *Let  $\mathbb{D} = \langle \{D^\sigma\}, \mathcal{A} \rangle$  be an order-extensional, continuous model of PCF following the standard interpretation. Then, for each PCF-type  $\sigma$ ,  $D^\sigma$  is a Scott domain. Further,  $\mathbb{D}$  is fully abstract if and only if every compact element of each value domain  $D^\sigma$  is definable.* ■

We divide this result into three propositions and prove each one of them in turn.

**Proposition 3.4.3.** *If  $\langle \{D^\sigma\}, \mathcal{A} \rangle$  is an order-extensional, continuous model of PCF following the standard interpretation, then for each type  $\sigma$ , the domain  $D^\sigma$  is a Scott domain. So, in particular, each  $D^\sigma$  is a strongly  $\omega$ -algebraic CPO.*

**Proof.** To prove this result, it suffices to show that for each type  $\sigma$ ,

- $D^\sigma$  has an inverse system  $\{\psi_n^\sigma : n \in \omega\}$  of projections,
- any two elements  $x, y$  of  $D^\sigma$  have a meet  $x \sqcap y$  in  $D^\sigma$ .

We show that such semantic objects as the projection function  $\psi_n^\sigma$  as well as the meet operation exist by exhibiting the PCF-terms

$$\Psi_n^\sigma : \sigma \Rightarrow \sigma \quad \text{glb}^\sigma : \sigma \Rightarrow (\sigma \Rightarrow \sigma)$$

that define them respectively.

Before we introduce these terms, note that the language PCF does not have a constant which tests for equality of two ground values of integer type, though such a test is definable in terms of the test-for-zero constant and the fixed-point operator. The idea is simple: given any two integers, decrement both in step until one of them reaches zero; at this point, check if the other is zero. More formally, consider the PCF-term  $\text{eq} : \iota \Rightarrow \iota \Rightarrow o$  defined as follows:

$$\begin{aligned} \mathbf{Y}^{\iota \Rightarrow \iota \Rightarrow o} & (\lambda f^{\iota \Rightarrow \iota \Rightarrow o}. \lambda x^\iota. \lambda y^\iota. \text{cond}^\iota (\text{l-or}(\text{zero}?x)(\text{zero}?y)) \\ & (\text{l-and}(\text{zero}?x)(\text{zero}?y))(f(\text{pred}x)(\text{pred}y))); \end{aligned}$$

where the PCF-terms  $\text{l-or}$  (“left or”) and  $\text{l-and}$  (“left and”) both of type  $o \Rightarrow o \Rightarrow o$  are defined as follows:



$$\begin{aligned}
\text{l-and} &\stackrel{\text{def}}{=} \lambda x^o. \lambda y^o. \text{cond}^o x (\text{cond}^o y \text{tf}) f, \\
\text{l-or} &\stackrel{\text{def}}{=} \lambda x^o. \lambda y^o. \text{cond}^o x t (\text{cond}^o y \text{tf}).
\end{aligned}$$

For the base types  $\iota$  and  $o$ , and each  $n \in \omega$ , we define

$$\begin{aligned}
\Psi_n^o &\stackrel{\text{def}}{=} \lambda x^o. x, \\
\Psi_0^\iota &\stackrel{\text{def}}{=} \lambda x^\iota. \text{cond}^\iota (\text{zero}? x) 0 \Omega, \\
\Psi_{n+1}^\iota &\stackrel{\text{def}}{=} \lambda x^\iota. \text{succ}(\Psi_n^\iota (\text{pred} x)), \\
\text{glb}^o &\stackrel{\text{def}}{=} \lambda x^o. \lambda y^o. \text{cond}^o x (\text{cond}^o y t \Omega) (\text{cond}^o y \Omega f), \\
\text{glb}^\iota &\stackrel{\text{def}}{=} \lambda x^\iota. \lambda y^\iota. \text{cond}^\iota (\text{eq} x y) x \Omega.
\end{aligned}$$

It is easy to see that for any  $m, n \geq 0$ ,  $\Psi_n^\iota m$  evaluates to  $n$  if  $m \geq n$ ; otherwise, it evaluates to  $m$ . Recursively, for any function type  $\sigma \Rightarrow \tau$ , we define:

$$\begin{aligned}
\Psi_n^{\sigma \Rightarrow \tau} &\stackrel{\text{def}}{=} \lambda x^{\sigma \Rightarrow \tau}. \lambda y^\sigma. \Psi_n^\tau (x (\Psi_n^\sigma y)), \\
\text{glb}^{\sigma \Rightarrow \tau} &\stackrel{\text{def}}{=} \lambda x^{\sigma \Rightarrow \tau}. \lambda y^{\sigma \Rightarrow \tau}. \lambda z^\sigma. \text{glb}^\tau (xz) (yz).
\end{aligned}$$

The  $n + 1$ -ary version of the term  $\text{glb}^\sigma$  is defined as:

$$\text{glb}_{n+1}^\sigma \stackrel{\text{def}}{=} \lambda x_1^o \cdots \lambda x_{n+1}^o. \text{glb}^\sigma (\text{glb}_n^\sigma x_1 \cdots x_n) x_{n+1}.$$

**Exercise** Verify that  $\{\Psi_n^\sigma\}$  and  $\{\text{glb}^\sigma\}$  define respectively a system of projections and the collection of meet operations for the value domains. ■

**Proposition 3.4.4 (Milner).** *Let  $\mathbb{D} = \langle \{D^\sigma\}, \mathcal{A} \rangle$  be an order-extensional, continuous model of PCF following the standard interpretation. If  $\mathbb{D}$  is fully abstract, then every compact element of each value domain  $D^\sigma$  is definable.*

**Proof.** We prove the implication by establishing its contraposition. Suppose there is a type  $\sigma$  and a natural number  $m$  such that some compact element  $\epsilon = \psi_m^\sigma(\epsilon)$  of  $D^\sigma$  is not definable. Since every compact element of a ground-type value domain  $D^\beta$  is definable by assumption,  $\sigma$  must be a function type. Let  $\sigma \equiv (\sigma_1, \dots, \sigma_n, \beta)$  be a “minimal” such type; that is to say, every compact element of  $D^{\sigma_i}$  is definable, for each  $1 \leq i \leq n$ .

Let  $K$  be the collection of precisely those elements of  $\psi_m^\sigma(D^\sigma)$  which are definable. We partition  $K = K^+ \cup K^-$  in the following way: define  $K^+ \stackrel{\text{def}}{=} \{d \in K : \epsilon \leq d\}$ . Since the set  $\psi_m^\sigma(D^\sigma)$  is finite, we may write  $K^+ = \{\gamma_i : 1 \leq i \leq p\}$  and  $K^- = \{\delta_i : 1 \leq i \leq q\}$ , for some  $p, q \geq 0$ . By assumption, there are terms  $\Gamma_i$  and  $\Delta_i$  which define the elements  $\gamma_i$  and  $\delta_i$  respectively. Now, the set  $K^+$  may or may not be empty. First, we consider the case of  $K^+ \neq \emptyset$ .



Write  $\Gamma \equiv \Psi_m^\sigma(\text{glb}_p^\sigma \Gamma_1 \cdots \Gamma_p)$ , and  $\gamma \stackrel{\text{def}}{=} \mathcal{A}[\Gamma](\perp)$ . We have

$$\gamma = \psi_m^\sigma(\gamma_1 \sqcap \cdots \sqcap \gamma_p) = \psi_m^\sigma(\gamma_1) \sqcap \cdots \sqcap \psi_m^\sigma(\gamma_p) = \gamma_1 \sqcap \cdots \sqcap \gamma_p.$$

Hence, we have  $\epsilon \leq \gamma$ , and  $\Psi_m^\sigma(\gamma) = \gamma$ , that is to say,  $\gamma \in K^+$ . Note that  $\epsilon < \gamma$  since  $\epsilon$  is not definable. Hence, by order-extensionality, there are elements  $\alpha_1 \in D^{\sigma_1}, \dots, \alpha_n \in D^{\sigma_n}$  such that  $\epsilon\alpha_1 \cdots \alpha_n < \gamma\alpha_1 \cdots \alpha_n$ . Moreover, by the assumption of the minimality of  $\sigma$ , there are PCF-terms  $A_1, \dots, A_n$  defining  $\alpha_1, \dots, \alpha_n$  respectively. Similarly, for  $1 \leq i \leq q$  and  $1 \leq j \leq n$ , there are PCF-terms  $B_j^i$  defining  $\beta_j^i$  such that for each  $1 \leq i \leq q$ ,  $\epsilon\beta_1^i \cdots \beta_n^i \not\leq \delta_i\beta_1^i \cdots \beta_n^i$ . We claim that the following is valid:

There is a base type  $\beta_0$ , and a compact element  $d_0 \in D^{\beta_0}$ , such that for any base type  $\beta$ , and for any compact element  $d \in D^\beta$ , the function  $(d \searrow d_0) : D^\beta \Rightarrow D^{\beta_0}$  defined as follows is PCF-definable: for any  $x \in D^\beta$ ,

$$(d \searrow d_0)(x) \stackrel{\text{def}}{=} \begin{cases} d_0 & \text{if } x \geq d, \\ \perp & \text{otherwise.} \end{cases}$$

We write  $[(d \searrow d_0)]$  for a PCF-term that defines the element  $(d \searrow d_0)$ .

For example, for  $\beta_0 = o$ , we have

$$\begin{aligned} [(\perp \searrow t)] &= \lambda x^o. t, \\ [(n \searrow t)] &= \lambda x^t. \text{cond}^o(\text{eq} x n) t \Omega, \\ [(t \searrow t)] &= \lambda x^o \text{cond}^o x t \Omega, \\ [(f \searrow t)] &= \lambda x^o \text{cond}^o x \Omega t. \end{aligned}$$

Consider the two terms of type  $\sigma \Rightarrow \beta_0$ :

$$\begin{aligned} \Xi_1 &\stackrel{\text{def}}{=} \lambda x^\sigma. \text{glb}_{q+1}^{\beta_0} ([(\epsilon\beta_1^1 \cdots \beta_n^1 \searrow d_0)](\Psi_m^\sigma x B_1^1 \cdots B_n^1)) \\ &\quad \vdots \\ &\quad ([(\epsilon\beta_1^q \cdots \beta_n^q \searrow d_0)](\Psi_m^\sigma x B_1^q \cdots B_n^q)) \\ &\quad ([(\epsilon\alpha_1^q \cdots \alpha_n^q \searrow d_0)](\Psi_m^\sigma x A_1 \cdots A_n)); \end{aligned}$$

$$\begin{aligned}
\Xi_2 &\stackrel{\text{def}}{=} \lambda x^\sigma. \text{glb}_{q+1}^{\beta_0} ([(\epsilon\beta_1^1 \cdots \beta_n^1 \searrow d_0)](\Psi_m^\sigma x B_1^1 \cdots B_n^1)) \\
&\quad \vdots \\
&\quad ([(\epsilon\beta_1^q \cdots \beta_n^q \searrow d_0)](\Psi_m^\sigma x B_1^q \cdots B_n^q)) \\
&\quad ([(\gamma\alpha_1^q \cdots \alpha_n^q \searrow d_0)](\Psi_m^\sigma x A_1 \cdots A_n)).
\end{aligned}$$

It is straightforward to see that  $\mathcal{A}[\Xi_1](\perp)\epsilon = d_0$  whereas  $\mathcal{A}[\Xi_2](\perp)\epsilon = \perp$ . However, we will now show that  $\Xi_1 \approx \Xi_2$ . By the Context Lemma and the Adequacy Theorem, it suffices to prove  $\mathcal{A}[\Xi_1 N](\perp) = \mathcal{A}[\Xi_2 N](\perp)$  for any closed term  $N$  of type  $\sigma$ . Note that  $\psi_m^\sigma(\mathcal{A}[N](\perp)) \in K$ . There are two cases:

- $\mathcal{A}[\Psi_m^\sigma N](\perp) \in K^+$ : we have  $\mathcal{A}[\Xi_1 N](\perp) = \mathcal{A}[\Xi_2 N](\perp) = d_0$ ;
- $\mathcal{A}[\Psi_m^\sigma N](\perp) \in K^-$ : in which case,  
 $\mathcal{A}[\Xi_1 N](\perp) = \mathcal{A}[\Xi_2 N](\perp) = \perp$ .

Next, we consider the case of  $K^+ = \emptyset$ . The same result can be achieved with the terms

$$\begin{aligned}
\Xi_1 &\stackrel{\text{def}}{=} \lambda x^\sigma. \text{glb}_q^{\beta_0} ([(\epsilon\beta_1^1 \cdots \beta_n^1 \searrow d_0)](\Psi_m^\sigma x B_1^1 \cdots B_n^1)) \\
&\quad \vdots \\
&\quad ([(\epsilon\beta_1^q \cdots \beta_n^q \searrow d_0)](\Psi_m^\sigma x B_1^q \cdots B_n^q));
\end{aligned}$$

$$\Xi_2 \stackrel{\text{def}}{=} \Omega.$$

**Proposition 3.4.5.** *Let  $\mathbb{D} = \langle \{D^\sigma\}, \mathcal{A} \rangle$  be an order-extensional, continuous model of PCF. If for every type  $\sigma$ , each compact element of the value domain  $D^\sigma$  is definable, then the model  $\mathbb{D}$  is fully abstract.*

**Proof.** We show: for any type  $\sigma \equiv (\sigma_1, \dots, \sigma_n, \beta)$ , and for any PCF-terms  $M$  and  $N$  of type  $\sigma$ , given any program context  $C[X]$  such that  $X$  is of type  $\sigma$  and that both  $C[M]$  and  $C[N]$  are programs, then  $\mathcal{A}[M] \subseteq \mathcal{A}[N]$  if and only if  $\mathcal{A}[C[M]] \subseteq \mathcal{A}[C[N]]$ . Suppose  $\{x_1, \dots, x_m\}$  is the set of free variables of  $M$  or  $N$ . The implication “ $\Rightarrow$ ” follows immediately from the continuity of the interpretation of  $C[X]$ . For the other direction “ $\Leftarrow$ ”, it suffices to show that for any environment  $\rho$  such that  $\rho(x_j)$  is compact for each  $1 \leq j \leq m$ , and for any  $d_i \in K(D^{\sigma_i})$  with  $1 \leq i \leq n$ ,

$$\mathcal{A}[M](\rho)d_1 \cdots d_n \subseteq \mathcal{A}[N](\rho)d_1 \cdots d_n.$$

For each  $1 \leq j \leq m$ , let  $P_j$  define  $\rho(x_j)$ ; and for  $1 \leq i \leq n$ , let  $Q_i$  define  $d_i$ . Let  $C[X] \equiv (\lambda x_1 \cdots x_m. X)P_1 \cdots P_m Q_1 \cdots Q_n$ . We then have

$$\begin{aligned}
\mathcal{A}[M](\rho)d_1 \cdots d_n &= \mathcal{A}[(\lambda \vec{x}.M)\vec{P}\vec{Q}](\perp) \\
&= \mathcal{A}[C[M]](\perp) && \text{by assumption} \\
&\leq \mathcal{A}[C[N]](\perp) \\
&= \mathcal{A}[(\lambda \vec{x}.N)\vec{P}\vec{Q}](\perp) \\
&= \mathcal{A}[N](\rho)d_1 \cdots d_n.
\end{aligned}$$

■

### 3.5 Full abstraction and non-full abstraction results

In [Plotkin, 1977], Plotkin proved that the standard Scott-continuous function space model is not fully abstract for PCF, but it is for PCF extended by a “parallel-conditional” construct. We first present a proof of the non-full abstraction result by appealing to the syntactic stability property of PCF. We prove full abstraction using a different extension of PCF: PCF extended by “parallel or”. This proof is due to Curien [Curien, 1993a].

**Proposition 3.5.1.** *The standard continuous function space model for PCF is not fully abstract.*

**Proof.** Write  $\text{Bool}_\perp$  for the standard flat CPO of booleans. Let  $\text{por}$  be the continuous function from  $\text{Bool}_\perp \times \text{Bool}_\perp$  to  $\text{Bool}_\perp$  defined by

$$\text{por}(b_1, b_2) = \begin{cases} \text{t} & \text{if } b_1 = \text{t} \text{ or } b_2 = \text{t}, \\ \text{f} & \text{if } b_1 = b_2 = \text{f}, \\ \perp & \text{otherwise.} \end{cases}$$

Note that  $\text{por}$  is a compact element of the standard model. By the Stability Theorem, the function  $\text{por}$  is not PCF-definable. The result then follows from the Definability Theorem. See [Plotkin, 1977] or [Gunter, 1992, pp. 179–182] for another proof. ■

**Two approaches to full abstraction** As the preceding proof clearly indicates, the standard model is not fully abstract for PCF because it is too rich a structure for the language. This mismatch arises because on the one hand, there are Scott-continuous functions like parallel or which can only be calculated by parallel algorithms; on the other, PCF is sequential (see Section 1). This observation was first made by Scott in [Scott, 1969]. There are two ways by which we may redress the situation and try to achieve full abstraction. The first approach fixes the model and upgrades the language. The second approach regards the language as prior and cuts the model down to size. We shall call the first the *expansive approach* and the second the *restrictive approach*.

**An expansive approach: PCF with “parallel-or”: PCFP** We extend PCF with a parsimonious parallel construct called “parallel-or”. Though the construct has a very simple operational behaviour, such an addition renders the augmented language non-sequential (say, in the Plotkin-Kahn sense). Since PCF is sequential, the construct does add something to the language. In fact, Curien shows that a simple parallel construct like parallel-or adds enough expressive power to PCF so that all compact elements of the standard Scott-continuous function space model are definable in the extended language. By the Definability Theorem, the standard model is fully abstract for the extended language.

**Parallel-or** We consider a new language PCFP which is obtained from PCF by augmenting it by a first-order parallel-or constant, p-or of type  $o \Rightarrow o \Rightarrow o$ . For any PCFP-term  $B$  of type  $o$ ,

$$\begin{aligned} \text{p-or } t \ B &> \ t \\ \text{p-or } B \ t &> \ t \\ \text{p-or } f \ f &> \ f. \end{aligned}$$

For each  $n \geq 2$ , we define the  $n$ -ary *parallel-or* constant recursively as

$$\text{p-or}_{n+1} \equiv \lambda x_1^o \cdots x_{n+1}^o. \text{p-or}(\text{p-or}_n x_1 \cdots x_n) x_{n+1}.$$

We extend the notion of standard interpretation to include the interpretation of parallel-or in the following way: for  $b_1, b_2 \in \mathbb{B}_\perp$ ,

$$\mathcal{A}[\text{p-or}](\perp) b_1 b_2 = \begin{cases} t & \text{if } b_1 = t \text{ or } b_2 = t, \\ f & \text{if } b_1 = b_2 = f, \\ \perp & \text{otherwise.} \end{cases}$$

Define a “negation” term neg of type  $o \Rightarrow o$ , “parallel-and” p-and and “left-and” l-and both of type  $o \Rightarrow o \Rightarrow o$  as follows:

$$\begin{aligned} \text{neg} &\equiv \lambda x^o. \text{cond}^o x f t, \\ \text{p-and} &\equiv \lambda x^o. \lambda y^o. \text{neg}(\text{p-or}(\text{neg } x)(\text{neg } y)), \\ \text{l-and} &\equiv \lambda x^o. \lambda y^o. \text{cond}^o x(\text{cond}^o y t f). \end{aligned}$$

Next, we define the  $n$ -ary “parallel-and” p-and <sub>$n+1$</sub>  as

$$\lambda x_1^o \cdots x_{n+1}^o. \text{p-and}(\text{p-and}_n x_1 \cdots x_n) x_{n+1}.$$

Similarly, we extend l-and to an  $n$ -ary “left-and” l-and <sub>$n$</sub> .

We recall an important domain-theoretic fact:

**Lemma 3.5.2 (Scott).** *If  $D$  and  $E$  are Scott domains, then so is the CPO  $[D \Rightarrow E]$  of continuous functions from  $D$  to  $E$ , ordered extensionally. Further, its compact elements are all least upper bounds of finite sets of elements of the form  $(d \searrow e)$ , with  $d$  and  $e$  ranging over compact elements of  $D$  and  $E$  respectively.* ■

For each Scott domain  $D$ , we use  $K(D)$  to denote the collection of compact elements of  $D$ . Fix a type  $\sigma \equiv (\sigma_1, \dots, \sigma_n, \beta)$ . By the above lemma, it is an easy exercise to see that any compact element of  $D^\sigma$  is just  $\bigsqcup F$  where  $F$  ranges over finite subsets of  $D^\sigma$  satisfying the following conditions:

- (1) each element of  $F$  has the shape  $(a_1 \searrow (a_2 \searrow \dots (a_n \searrow v) \dots))$  such that  $a_i \in K(D^{\sigma_i})$  for each  $1 \leq i \leq n$ , and that  $v \in K(D^\beta)$  with  $v \neq \perp$ ,
- (2) for any two elements

$$(a_1 \searrow (a_2 \searrow \dots (a_n \searrow v) \dots))$$

and

$$(a'_1 \searrow (a'_2 \searrow \dots (a'_n \searrow v') \dots))$$

of  $F$ , whenever  $a_i \uparrow a'_i$  for each  $1 \leq i \leq n$ , then  $v = v'$ .

In fact, a set satisfying the first of the above conditions also satisfies the second if and only if  $F$  is consistent as a subset of  $D^\sigma$ . Following Gunter, we call such a set  $F$  *crisp* just in case  $F$  satisfies conditions (1) and (2).

**Theorem 3.5.3 (Plotkin).** *The standard continuous model is order- extensional and fully abstract for PCFP.*

We appeal to the Definability Theorem and prove definability of compact elements by induction on the structure of types. In this case, it is necessary to work with a strengthened hypothesis. It suffices to prove that each of the following assertions is valid for every type  $\sigma$ :

- (1) for each  $a \in K(D^\sigma)$ , there is a PCFP-term  $M[a]$  defining  $a$ ,
- (2) for each  $a \in K(D^\sigma)$ , there is a PCFP-term  $T[a]$  defining  $(a \searrow \mathbf{t})$ ,
- (3) for each  $a, b \in K(D^\sigma)$ , if  $a$  and  $b$  are incompatible, then there is a PCFP-term  $D[a, b]$  defining  $(a \searrow \mathbf{t}) \sqcup (b \searrow \mathbf{f})$ .

Our strategy is to prove the above assertions simultaneously by structural induction on the type  $\sigma$ .

(1): The base case corresponding to ground types  $\sigma$  and  $\iota$  is straightforward, and so is omitted. For the inductive case of type  $\sigma \equiv (\sigma_1, \dots, \sigma_n, \beta)$ , consider any compact element  $a \in K(D^\sigma)$ . We argue by induction on the least number  $m$  such that there is a crisp set  $F = \{f^j : 1 \leq j \leq m\}$  such that  $a = \bigsqcup F$ . We call  $m$  the breadth of  $a$ . Let  $\mathbf{H}(m)$  be the statement that every compact element of  $D^\sigma$  of breadth  $m$  is definable. If the number



$m$  is zero, then  $F = \emptyset$ , and  $a$  is just  $\perp$ . We prove the inductive case of  $\mathbf{H}(m)$  for  $m \geq 1$ . We write each

$$f^j = (a_1^j \searrow (a_2^j \searrow \cdots (a_n^j \searrow v^j) \cdots)).$$

There are two subcases (modulo permutations of  $\{f^1, \dots, f^m\}$ ):

(1a)  $v^1 = v^2 = \cdots = v^m$ ,

(1b) there is some number  $k$  with  $1 \leq k < m$  such that  $v^1 = \cdots = v^k$ , and that for each  $k+1 \leq j \leq m$ ,  $v^1 \neq v_j$ .

In the case of (1a), by the induction hypotheses (1) and (2) respectively, there are PCFP-terms  $V$  and  $T_i^j \equiv T[a_i^j]$  defining  $v_1$  and  $(a_i^j \searrow \mathbf{t})$  respectively, for each  $1 \leq i \leq n$  and  $1 \leq j \leq m$ . It is easy to see that

$$\begin{aligned} M[a] &\equiv \lambda x_1^{\sigma_1} \cdots x_n^{\sigma_n} . \text{cond}^\beta [\text{p-or}_m (\text{l-and}_n (T_1^1 x_1) \cdots (T_n^1 x_n)) \\ &\quad \cdots (\text{l-and}_n (T_1^m x_1) \cdots (T_n^m x_n))] V \Omega. \end{aligned}$$

In the case of (1b), since  $F$  is a crisp set, for each pair  $(u, v)$  where  $1 \leq u \leq k$  and  $k+1 \leq v \leq m$ , there is an  $i$  (depending on  $(u, v)$ ) such that  $a_i^u$  and  $a_i^v$  are incompatible. Applying the induction hypothesis (3), there is a PCFP-term  $D^{u,v} \equiv D[a_i^u, a_i^v]$  defining  $(a_i^u \searrow \mathbf{t}) \sqcup (a_i^v \searrow \mathbf{f})$ . Applying the induction hypotheses  $\mathbf{H}(k)$  and  $\mathbf{H}(m-k)$  respectively, there are PCFP-terms  $M_1$  and  $M_2$  defining the compact elements  $f^1 \sqcup \cdots \sqcup f^k$  and  $f^{k+1} \sqcup \cdots \sqcup f^m$  respectively. It remains to prove that

$$M[a] \stackrel{\text{def}}{=} \lambda x_1^{\sigma_1} \cdots x_n^{\sigma_n} . \text{cond}^\beta (D x_1 \cdots x_n) (M_1 x_1 \cdots x_n) (M_2 x_1 \cdots x_n)$$

defines  $a$  provided there is a PCFP-term  $D$  satisfying the following properties:

For each  $i$ , and for any  $d_i \in D^{\sigma_i}$ , if  $ad_1 \cdots d_n \neq \perp$ , then

- if  $ad_1 \cdots d_n = v_1$ , then  $\mathcal{A}[D](\perp) d_1 \cdots d_n = \mathbf{t}$ ,
- if  $ad_1 \cdots d_n \neq v_1$ , then  $\mathcal{A}[D](\perp) d_1 \cdots d_n = \mathbf{f}$ .

Consider the term

$$\begin{aligned} D &\equiv \lambda x_1^{\sigma_1} \cdots x_n^{\sigma_n} . \text{p-or}_k (\text{p-and}_{m-k} (D^{1,k+1} x_1 \cdots x_n) \cdots (D^{1,m} x_1 \cdots x_n)) \\ &\quad \vdots \\ &\quad (\text{p-and}_{m-k} (D^{k,k+1} x_1 \cdots x_n) \cdots (D^{k,m} x_1 \cdots x_n)). \end{aligned}$$

For  $p \geq 2$ , recall that the PCFP-term  $\text{p-and}_p$  has type  $\underbrace{o \Rightarrow o \Rightarrow \cdots \Rightarrow o}_{p+1}$  and has denotation satisfying the following property: for  $b_i \in \mathbb{B}_\perp$ ,

$$\mathcal{A}[\text{p-and}_p](\perp)b_1 \cdots b_p = \begin{cases} \mathbf{t} & \text{if } b_i = \mathbf{t} \text{ for every } i \leq p, \\ \mathbf{f} & \text{if } b_i = \mathbf{f} \text{ for some } i \leq p, \\ \perp & \text{otherwise;} \end{cases}$$

and similarly for  $\text{p-or}_k$ . To verify that  $D$  satisfies the necessary property, for any  $d_i \in D^{\sigma_i}$ , suppose  $ad_1 \cdots d_n = v_1$ . Then, there must be some  $u_0$  with  $1 \leq u_0 \leq k$  such that  $a_i^{u_0} \sqsubseteq d_i$  for every  $i$ . For this reason,

$$\mathcal{A}[\text{p-and}_{m-k}(D^{u_0, k+1}d_1 \cdots d_n) \cdots (D^{u_0, m}d_1 \cdots d_n)](\perp) = \mathbf{t};$$

and so  $\mathcal{A}[D](\perp)d_1 \cdots d_n = \mathbf{t}$ . Now, suppose  $ad_1 \cdots d_n \neq v_1$  and  $\neq \perp$ . Since  $D^\beta$  is a flat CPO, there must be some  $v_0$  such that  $k+1 \leq v_0 \leq m$  such that  $a_i^{v_0} \sqsubseteq d_i$  for every  $i$ . Hence, for every  $u$  such that  $1 \leq u \leq k$ ,  $\mathcal{A}[D^{u, v_0}](\perp)d_1 \cdots d_n = \mathbf{f}$ ; and so  $\mathcal{A}[D](\perp)d_1 \cdots d_n = \mathbf{f}$ .

(2) For simplicity, we may write  $a = (a^1 \searrow \bar{a}^1) \sqcup \cdots \sqcup (a^p \searrow \bar{a}^p)$ . By the induction hypotheses (1) and (2) respectively, there are PCFP-terms  $M[a^j]$  and  $T[\bar{a}^j]$  defining compact elements  $a^j$  and  $(\bar{a}^j \searrow \mathbf{t})$  respectively. The following term defines  $(a \searrow \mathbf{t})$ :

$$\lambda x^\sigma. \text{l-and}_m(T[\bar{a}^1](xM[a^1])) \cdots (T[\bar{a}^p](xM[a^p])).$$

(3) We write  $a$  as in the previous case, and  $b = (b^1 \searrow \bar{b}^1) \sqcup \cdots \sqcup (b^q \searrow \bar{b}^q)$ . Since  $a$  and  $b$  are incompatible, there are  $u$  and  $v$  such that  $(a^u \searrow \bar{a}^u)$  and  $(b^v \searrow \bar{b}^v)$  are incompatible; that is to say,  $a^u$  and  $b^v$  are compatible, and  $\bar{a}^u$  and  $\bar{b}^v$  are not. Applying induction hypothesis (3),  $D[\bar{a}^u, \bar{b}^v]$  is a term that defines  $(\bar{a}^u \searrow \mathbf{t}) \sqcup (\bar{b}^v \searrow \mathbf{f})$ . Similarly,  $M[a^u \sqcup b^v]$ ,  $T[a]$  and  $T[b]$  define  $a^u \sqcup b^v$ ,  $(a \searrow \mathbf{t})$  and  $(b \searrow \mathbf{t})$  respectively. Hence, the following term

$$D[a, b] \equiv \lambda x^\sigma. \text{cond}^\beta(D[\bar{a}^u, \bar{b}^v](xM[a^u \sqcup b^v]))(T[a]x)(\text{neg}(T[b]x))$$

defines  $(a \searrow \mathbf{t}) \sqcup (b \searrow \mathbf{f})$ . This concludes our proof of the definability result and hence the full abstraction result.  $\blacksquare$

**Remark** The full abstraction of the Scott-continuous function space model with respect to PCFP is due independently to Curien and Abramsky, and the preceding argument is based on Curien's proof in [Curien, 1993a]. In [Plotkin, 1977], Plotkin extended PCF with a *parallel-conditional* constant  $\text{pif}^\sigma$  to achieve full abstraction of the language with respect to the standard Scott-continuous function space model. The one-step transition semantics of the construct  $\text{pif}^\beta : \sigma \Rightarrow \beta \Rightarrow \beta$  is as follows:

$$\begin{aligned}
\text{pif}^\beta BMM &\rightarrow M \\
\text{pif}^\beta \mathbf{t}MN &\rightarrow M \\
\text{pif}^\beta \mathbf{f}MN &\rightarrow N.
\end{aligned}$$

The behaviour of the construct may be succinctly characterized semantically. We say that an interpretation of parallel-conditional is standard if the following property is satisfied: for any  $b \in \mathbb{B}_\perp$ , and for  $d_1, d_2 \in \mathbb{N}_\perp$ ,

$$\mathcal{A}[\text{pif}^c](\perp)bd_1d_2 \stackrel{\text{def}}{=} \begin{cases} d_1 & \text{if } b = \mathbf{t}, \\ d_2 & \text{if } b = \mathbf{f}, \\ d_1 & \text{if } b = \perp \text{ and } d_1 = d_2, \\ \perp & \text{otherwise.} \end{cases}$$

Parallel-or may be simulated using parallel-conditional by the term

$$\lambda x^\circ. \lambda y^\circ. \text{pif}^\circ xty.$$

In fact, Stoughton showed that parallel-or and parallel-conditional are inter-definable [Stoughton, 1991b].

Since the standard model does not characterize PCF, is there any model that does? More precisely, is there an order-extensional, continuous, fully abstract model for PCF? This question was answered by Milner [Milner, 1977] who constructed one such model and showed that it is unique up to isomorphism.

**Theorem 3.5.4 (Milner).** *There is a unique (up to isomorphism) inequationally fully abstract, order-extensional, continuous model of PCF. ■*

Milner constructed the unique fully abstract model for a combinatory logic version of PCF. His model is essentially a term model; the construction was based on suitable completions of sets of equivalence classes of terms in a way which is reminiscent of the construction of SFP domains (see [Plotkin, 1976]).

**A restrictive approach** In [Mulmuley, 1986; Mulmuley, 1987], Mulmuley gave a construction of the unique fully abstract model (as identified by Milner) by a mixture of semantic and syntactic means. His starting point was the Scott-continuous functions space model built up from the standard lattice interpretation of the numerals and booleans.

- First, a family of logical relations  $d \leq M$  indexed by types is defined between elements of the standard model and closed PCF-terms as follows:

$$\begin{array}{llll}
\text{ground type} & d \sqsubseteq M & \iff & d \sqsubseteq \mathcal{A}[M] \\
\text{function type } \sigma \Rightarrow \tau & d \sqsubseteq M & \iff & e \sqsubseteq M \implies de \sqsubseteq MN \\
& & & \text{for every } e \in D_\sigma \text{ and } N : \sigma.
\end{array}$$

- For each type  $\sigma$ , a preorder  $\leq$  is defined over the domain  $D_\sigma$  in the following way:

$$d \leq e \iff d \sqsubseteq M \implies e \sqsubseteq N \text{ for every term } M.$$

- The standard model is quotiented by the equivalence associated with the preorder. Care has to be taken in showing that the resultant poset is complete.

Because every value domain  $D_\sigma$  has a top element, the quotients may be turned into a retraction. In this way, a fully abstract model was constructed from which it is just a matter of lopping the top element off to arrive at Milner's unique fully abstract model.

**Refinements** We conclude this subsection by mentioning some recent refinements of the “classical” results of Plotkin and Milner by Stoughton [Stoughton, 1991c; Stoughton, 1991a]:

- There is only one extensional, continuous model for PCFP; it is precisely the unique fully abstract model.
- There is an equationally fully abstract model for PCFP distinct from the unique order-extensional, continuous, inequationally fully abstract model.

More recently, Sieber [Sieber, 1992] used the machinery of logical relations to characterize PCF-terms up to order 3. He introduced a collection of *sequential* logical relations over continuous models of PCF and showed that every PCF-definable element is invariant over these relations. Starting from a continuous model, one can define a new model by first throwing out those elements which are not invariant over the sequential logical relations and then by performing the standard extensional collapse. The main result of his work is that the resultant model is fully abstract for PCF-terms of order not greater than 3. The proof depends on a definability result for elements of order less than or equal to 2.

## 4 Towards a characterization of PCF-sequentiality

Though there is not yet a consensus that PCF is the canonical example of a (higher-type) sequential programming language, there is compelling evidence (much of which we have already come across in this chapter) that PCF computes only sequential functions. Of course, the adjective “sequential” has just been used in an informal sense. Expressed in technologically



more precise terms, the general intuition is that a language is sequential “if it can be implemented without time-slicing among multiple threads of control”, a useful formulation due to Cartwright, Curien and Felleisen. Part of the difficulty in making the above statement precise is that the notion of higher-type sequentiality is a slippery one. In our view, the definition of sequential computable functions at higher type poses the same kind of epistemological difficulties as that of the “effectively computable” functions. Although the idea of sequentiality is intuitively clear, it is very hard to formulate in precise, mathematical terms. As we shall see later, first-order sequentiality is quite easily characterized. The real difficulty is to define sequentiality at higher type.

It seems clear enough that PCF *qua* programming language *defines* a model of sequential computable functions at higher type. Let us call this model of sequential computation (formally defined in terms of the collection of higher-type numeric functions which are PCF-definable) *PCF-style sequentiality* or simply *PCF-sequentiality*. Now a fully abstract model of PCF which satisfies the twin criteria of abstract and synthetic description would be a good characterization of this particular model of higher-type sequentiality. What is less clear at this stage is whether PCF-sequentiality thus defined is in any sense *canonical*. Put differently, is PCF-sequentiality universal, or in any sense inevitable? Or is it just an *ad hoc* notion? Is there any evidence to support the following proposition which may be called the Church-Turing thesis for higher-type sequentiality?

**Church-Turing thesis for sequentiality** *The intuitively clear but informally defined collection of computable sequential functions at higher type are just the higher-type numeric functions which are PCF-definable.* ■

Like the Church-Turing Thesis, this thesis clearly does not admit of any mathematical proof in the conventional sense. One way to argue for its acceptance, again taking the cue from the Church-Turing Thesis, is to establish a number of intuitively appealing models of higher-type sequential computations of vastly distinct conceptions, each of which computes precisely the class of PCF-definable functions. This would make an interesting direction of research.

**Towards a solution** Various attempts at solving the full abstraction problem for PCF have faltered on a sticking point: there is an apparent tradeoff between *sequentiality* on the one hand, and *extensionality* on the other. Families of functions which make good computational sense and enjoy good closure properties at higher type (*e.g.* Scott-continuous functions and Berry’s stable functions) inevitably include some functions which can only be computed by parallel algorithms; whereas mathematical structures which are intuitively sequential are invariably too intensional in nature to behave properly as a (cartesian closed) category of functions. This point



will be developed further in this section.

The rest of this section may be seen as a systematic account of the major attempts at solving the full abstraction problem (in the sense that we have made precise) from the late 1970's to the present. The only book-length treatment of the material covered in this section is the second edition of Curien's book [Curien, 1993a], and we recommend it.

### Continuous models of PCF and related languages: a summary of properties

Models	CCC	Ext.	Order-ext.	Fully abstract language
Scott-continuous function	Yes	Yes	Yes	PCFP
dI-domains + stable function	Yes	Yes	No	?
CDSS + sequential algorithms	Yes	No	No	<b>CDS</b> , also PCF + catch
CDSS + observable algorithms	Yes	Yes	Yes	SPCF
dI-domains + strongly stable fn.	Yes	Yes	No	?

#### 4.1 Concrete data structures and sequential functions

There is a very natural description of first-order sequential functions, *i.e.* functions which take tuples of ground values, say natural numbers, as arguments and produce ground values as results. We can think of such first-order functions as functions from a product of flat CPOs to a CPO. The

intuition is that the argument has to be visited in a specific order: if the function is strict and if it is not the constantly bottom function, then there is a component, say the  $i$ -th component  $x_i$ , which is needed at the start of the computation and it is evaluated by the algorithm that computes the function in question. If  $x_i$  is undefined, then so is the computation; otherwise, the computation then proceeds by evaluating, say, the  $j$ -th component of the argument —  $j$ , in general, depends on  $i$ ; and the computation goes on. So, for example, in the sense of Milner [Milner, 1977], a continuous function from CPOs  $D_1 \times \cdots \times D_{n+1}$  to  $E$  is *sequential* if either it is a constant, or there is an integer  $i$  with  $1 \leq i \leq n+1$  such that  $f$  is strict in the  $i$ -th argument, and the function obtained by fixing the  $i$ -th argument is sequential. Other attempts at defining first-order sequentiality have also been made by Vuillemin [Vuillemin, 1974] and Sazonov [Sazonov, 1975].

**Description of sequentiality: some requirements** These definitions of sequentiality have some drawbacks. First, they apply to functions with two or more arguments. (For a function with one argument, the notion reduces to one of strictness.) Secondly, the formulations capture the intuition of sequentiality only when the domains involved are flat CPOs; they break down at higher types. Nevertheless, they give us a good indication of some of the requirements for a description of sequentiality, whether higher-type or not:

- *Local partiality.* Neededness or strictness is a first (if crude) approximation to sequentiality. Any theory of sequentiality must give expression to *local partiality* (and undefinedness).
- *State.* Sequentiality describes a computational process “extended over time”: it is an intensional property. Any concrete analysis of sequentiality is relative to a notion of *state* of a computation.
- *Place.* Naïvely, to compute sequentially is to do one thing at a time. So a notion of *place* indicating different possible sites of subcomputations (so that it makes sense to speak of “doing *many* things but one thing at a time”) is essential for articulating any notion of sequentiality. We have just seen Milner’s definition of first-order sequentiality which uses the component of a cartesian product to express the notion of place as a first approximation.

Kahn and Plotkin’s concrete data structure (CDS) is just what the name indicates: it is a concrete (or better, intensional) mathematical structure well-suited for denoting computational data structure. The main conceptual contribution of Kahn and Plotkin’s work is a truly innovative formulation of the notions of local partiality, state and place which form the basis of an abstract notion of sequentiality that extends to higher types. We shall see later that the corresponding notion of place in the Kahn-Plotkin setting is that of a *cell*. Cells may or may not be filled with *values*, thus

giving expression to a notion of local partiality. The notion of the state of a computation is an essential part of the definition of a CDS.

To motivate the definition which may look rather complicated at first sight, it is helpful to consider an example of CDS: formal (partial) terms regarded as labelled trees over a given first-order signature  $\Sigma$ . Such tree-like terms are often represented as partial functions from the set  $\mathbb{N}^*$  of finite sequences of natural numbers to  $\Sigma$ . For example, the term  $f(a, g(b, \Omega), \Omega)$ , whose representation as a labelled tree is shown in Figure 3,

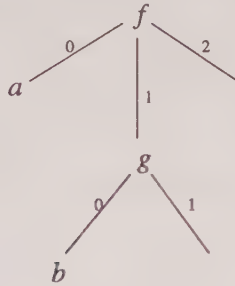


Fig. 3. The term  $f(a, g(b, \Omega), \Omega)$  represented as a labelled tree.

may equivalently be represented by the following partial function from  $\mathbb{N}^*$  to  $\Sigma$ :

$$\left\{ \begin{array}{ll} \epsilon & \mapsto f, \\ 0 & \mapsto a, \\ 1 & \mapsto g, \\ 1 \cdot 0 & \mapsto b. \end{array} \right.$$

For our purpose, it is more convenient to represent partial functions as graphs, so the above function corresponds to the set

$$\{ (\epsilon, f), (0, a), (1, g), (1 \cdot 0, b) \}$$

of pairs.

Given a first-order signature  $\Sigma$ , a partial  $\Sigma$ -term  $t$  is a set of pairs of the form  $(u, f)$  where  $u$  is just a finite sequence of natural numbers, and  $f \in \Sigma$  such that the following conditions are satisfied:

- *consistency*: whenever  $(u, f), (u, g) \in t$ , then  $f = g$ ;
- *safety*: if  $(u, f) \in t$  and  $u = v \cdot i$ , then  $(v, g) \in t$  for some  $g$  with arity greater than  $i$ .

The first component  $u$  in the pair  $(u, f)$  is just the *occurrence* of the symbol  $f$  in the term  $t$ . Thus, the membership predicate  $(u, f) \in t$  may be read as: “the symbol  $f$  occurs in the term  $t$  at occurrence  $u$ ”, or more formally,  $t / u = f$ , as we have already seen in Section 2. The consistency condition ensures that the set of pairs defines the graph of a functional relation. The safety condition corresponds to the requirement that the tree construction respects arity of function symbols.

**Concrete data structure** The description of a partial  $\Sigma$ -term in terms of a set of pairs of the shape “(*occurrence, symbol*)” is an instructive example of a concrete data structure. Formally, a *concrete data structure*  $M$  is a structure  $\langle C, V, E, \vdash \rangle$  where

- $C$  is the set of *cells*,
- $V$  is the set of *values*,
- the collection  $E$  of events is a subset of  $C \times V$  (we write an event as  $(c, v)$ ),
- $\vdash$  is an *enabling relation* between finite collections of events on the one hand and cells on the other. For any finite subset  $X$  of  $E$  (written  $X \subseteq_{\text{fin}} E$ ), we read  $X \vdash c$  as “ $X$  enables the cell  $c$ ”, or “ $X \vdash c$  is an enabling”.

A CDS is an elaborate structure for denoting computations. It is more concrete than, say, the setting of CPOs. Therefore, it allows for a more “local” or lower-level presentation of computations. With reference to the preceding discussion on the three conceptual elements (namely, place, local partiality and state) for articulating sequentiality, in the CDS approach, the notion of cell corresponds to the requirement of place. Computation proceeds in a discrete manner; each such quantum step corresponds to an *event* which is the filling of a cell with a value. At any point during a computation, some cells may be filled; there may be cells which are not filled. The possibility of an unfilled cell gives expression to a notion of local partiality. The collection of cells which have been filled gives a “snap shot” of the computation at that point. This gives rise to a natural way of describing the state of a computation. Formally, a *state*  $x$  of  $M$  is a subset of  $E$  satisfying the following conditions:

- *consistency*: whenever  $(c, v) \in x$  and  $(c, v') \in x$ , then  $v = v'$ ;
- *safety*: whenever  $(c, v) \in x$ , then there is a sequence of event  $e_0, \dots, e_n$  such that  $e_n = (c, v)$ ,  $e_i = (c_i, v_i) \in x$ , and  $\{e_j : j < i\}$  contains an enabling of  $c_i$  for all  $i \leq n$ .

Whenever an event takes place (*i.e.* the filling of a cell) in the course of a computation, say at state  $x$ , there is a possible knock-on effect on cells. Some cells may become enabled (*i.e.* available for filling) as a result so that at the next step, any one of these cells may be filled. This priming effect

is captured by the enabling relation  $\vdash$ . This then, is Kahn and Plotkin's vision of sequential computation.

We introduce the following shorthand:

$$\begin{aligned} c \in \mathbf{F}(x) & \quad \exists v \in V. (c, v) \in x && \text{"}c \text{ is filled in } x\text{"}, \\ c \in \mathbf{E}(x) & \quad \exists X \subseteq_{\text{fin}} E. X \vdash c && \text{"}c \text{ is enabled in } x\text{"}, \\ c \in \mathbf{A}(x) & \quad c \in \mathbf{E}(x) - \mathbf{F}(x) && \text{"}c \text{ is accessible from } x\text{"}. \end{aligned}$$

A cell  $c$  which has been enabled but not yet filled at a state  $x$ , *i.e.* accessible from  $x$ , is a place to which control of the computation may move "at the next step". Note that there may be more than one cell which is accessible from  $x$ . The notion of an accessible cell is an important idea in the framework of CDS: it enables Kahn and Plotkin to define a new notion of sequential function between CDSS, about which more anon.

We shall consider CDSS satisfying the following additional properties:

- *stability*: for any state  $x$ , each cell enabled in  $x$  has a unique enabling,
- *well-foundedness*: the binary relation  $c \ll d$  over cells, defined by

$$c \ll d \quad \Longleftrightarrow \quad \text{there is some } X \text{ such that } c \in \mathbf{F}(X) \text{ and } X \vdash d,$$

is well-founded.

A well-founded and stable CDS is called a *deterministic concrete data structure* (DCDS). Further, a stable, well-founded CDS is said to be *filiform* if every enabling is effected by at most one event. It is helpful to think of the state of a filiform DCDS as a tree: the nodes are events, and the edges correspond to enablings. We write  $x \prec_c y$  just in case  $y = x \cup \{ (c, v) \}$ , for some value  $v$ , and we say that  $y$  *covers*  $x$ .

**Examples** (i) The standard flat CPO of natural numbers may be expressed as the following CDS:

- $C = \{ ? \}$ ,
- $V = \mathbb{N}$ ,
- $E = C \times V$ ,
- the enabling relation  $\vdash$  is just  $\emptyset \vdash ?$ .

(ii) Given a first-order signature  $\Sigma$ , we can express the collection of partial  $\Sigma$ -terms formally as a CDS:

- $C = \mathbb{N}^*$ ,
- $V = \Sigma$ ,
- the enabling relation is specified as follows:

$$(u, f) \vdash u \cdot i \quad \Longleftrightarrow \quad i \text{ is less than the arity of } f.$$



(iii) The filiform DCDs of the pure  $\lambda$ -terms with partial and infinite terms is defined by the following data:

- $C = \{0, 1, 2\}^*$ ,
- $V = \{\cdot\} \cup \text{Var} \cup \{\lambda x : x \in \text{Var}\}$ ,
- $E = C \times V$ .

The enabling relation is defined as

$$\begin{aligned} \emptyset &\vdash \epsilon, \\ (u, \lambda x) &\vdash u \cdot 0, \\ (u, \cdot) &\vdash u \cdot 1, u \cdot 2. \end{aligned}$$

For instance, the term  $(\lambda xy.y)x$  corresponds to the following state:

$$\{(\epsilon, \cdot), (1, \lambda x), (1 \cdot 0, \lambda y), (1 \cdot 0 \cdot 0, y), (2, x)\}.$$

We shall see later in the section that filiform DCDs form a full (cartesian closed) subcategory of the category of DCDs and sequential algorithms which is cartesian closed. Since flat domains are represented by filiform DCDs, this subcategory is sufficient for studying the semantics of PCF.

The first construction of CDS is that of product. We tabulate the relevant data in the case of filiform DCDs as follows:

	$M$	$M'$	$M \times M'$
Cells	$c$	$c'$	$c \cdot 1, c' \cdot 2$
Values	$v$	$v'$	$v, v'$
Events	$(c, v)$	$(c', v')$	$(c \cdot 1, v), (c' \cdot 2, v')$
Enabling	$(c_1, v_1) \vdash c$	$(c'_1, v'_1) \vdash c'$	$(c_1 \cdot 1, v_1) \vdash c \cdot 1, (c'_1 \cdot 2, v'_1) \vdash c' \cdot 2$

**A representation theorem** Concrete data structures may be used to give denotational semantics for (sequential) programming languages. For this purpose, the denotation of a program will be an appropriate state of a CDS. What sort of structure does the collection of states of an arbitrary CDS have? A very satisfactory answer has been obtained by Plotkin and Kahn [Kahn and Plotkin, 1978]. They showed that the collection of states of a CDS, ordered by inclusion, forms an  $\omega$ -algebraic CPO which satisfies four axioms. The axioms spell out the structure of the finite elements of the CPO which are “adjacent” to each other with respect to the order. For elements  $x, y$  of a poset  $\langle X, \leq \rangle$ , we write  $x \prec y$  (read “ $y$  covers  $x$ ”, or more suggestively “ $x$  is immediately below  $y$ ”) if and only if  $x < y$  and that for

any element  $z$ , whenever  $x < z \leq y$  then  $z = y$ . In other words,  $y$  covers  $x$  if  $x$  is less than  $y$  and there is no other element in between.

A concrete domain  $D$  is an  $\omega$ -algebraic CPO satisfying the following axioms: for any finite elements  $x, y, z$  of  $D$ ,

- (F) Any finite element dominates only finitely many elements of  $D$ .
- (C) Whenever  $x < y$  and  $x < z$ , and  $y$  and  $z$  are distinct but compatible, then  $y \sqcup z$  exists, and  $y < y \sqcup z$  and  $z < y \sqcup z$ .
- (R) Whenever  $[x, y] > [x, z]$  then  $y = z$ .
- (Q) Whenever  $z < y$ ,  $z < x$ , and  $x$  and  $y$  are incompatible, then there is a unique  $x'$  such that  $z < x' \leq y$  and  $x$  and  $x'$  are incompatible.

The last axiom is called (Q) because the elements  $z, y, x'$  and  $x$  trace out (vaguely!) the shape of the letter Q. The axiom (F) is very often called (I) as in dI-domains. The following representation theorem, due to Kahn and Plotkin, gives an abstract characterization of the states of an arbitrary CDS as a concrete domain.

**Theorem 4.1.1 (Representation).** *The set of states of a CDS, ordered by inclusion, forms a concrete domain. For any concrete domain  $D$ , there is a CDS  $M(D)$  such that the set of states of  $M(D)$  and  $D$  are isomorphic. ■*

The first half of the theorem is more or less routine to prove. The main task in proving the second half is to construct the CDS  $M(D)$ . The original construction of Kahn and Plotkin has been generalized and simplified by Winskel in his work on event structures; see his thesis [Winskel, 1980] and the comprehensive treatment in [Winskel, 1987]. A detailed discussion of the material on event structures is beyond the scope of this chapter.

**Kahn-Plotkin sequential functions** Let  $f$  be a function from the set of states of the CDS  $M = \langle C, V, E, \vdash \rangle$  to the set of states of the CDS  $M' = \langle C', V', E', \vdash' \rangle$ . For any state  $x$  of  $M$ , and any cell  $c'$  accessible from  $f(x)$ , the function  $f$  is said to be *sequential* for  $x$  at  $c'$  if exactly one of the following is valid:

- for any  $z \supseteq x$ ,  $c'$  remains accessible from  $f(z)$ ,
- there is a cell  $c$  accessible from  $x$  such that for any  $y \supseteq x$ , if  $c'$  is filled in  $f(y)$  then  $c$  must also be filled in  $y$ .

Such a cell  $c$  is called a *sequentiality index* of  $f$  for the state  $x$  at the cell  $c'$ . We say that a function  $f$  is *sequential* if it is sequential for  $x$  at any cell  $c$  accessible from  $f(x)$ , for any state  $x$ .

An important example of a sequential function can be found in Berry's syntactic sequentiality result which we have already come across in Section 2 (compare the definition with, for example, Theorem 2.6.1). Another example is the stable injection-projection pairs of Berry and Curien [Berry and Curien, 1982] who used them to define a notion of approximation between CDSS, or more precisely, between concrete domains via the Rep-

resentation Theorem. This notion formed the basis of their approach to the solution of domain equations as the least fixed point of a continuous function on CDSSs.

The framework of CDS and the Kahn-Plotkin sequential function is a major conceptual advance in understanding higher-type sequentiality. The notion of Kahn-Plotkin sequential function is so natural that if it “worked”<sup>11</sup> at higher types, it would be a strong contender for a definition of higher-type sequential computation. Unfortunately, this is not the case.

**Theorem 4.1.2.** *The category of DCDS and sequential functions is not cartesian closed.* ■

A proof can be found in [Curien, 1993a, pp. 209–211]. The argument relies on the following result: the category of DCDS and sequential functions is equivalent to the category of distributive concrete domains and sequential functions. The crucial step in the proof by contradiction consists in exhibiting distributive concrete domains  $D$  and  $E$  such that the putative exponentiation from  $D$  to  $E$  is not a concrete domain. The construction is based on Berry’s “Gustave function” which we have already seen in Section 2.

The search for a cartesian closed category of higher-type sequential functions became the focus of research. Historically, the research bifurcated at this juncture. The sticking point is an apparent tradeoff between higher-type sequentiality which is inherently intensional, and functionality which is an extensional notion.

- One major effort consisted in relaxing the requirements of sequentiality but staying within the framework of functions. This led Berry to the notion of *stability*. The appropriate ordering between such functions, called stable functions, is not the standard extensional ordering but another ordering, called stable ordering.
- The other approach holds on to the central ideas behind CDS and Kahn-Plotkin sequential functions but sacrifices extensionality. Thus, Berry and Curien introduced *sequential algorithms* over CDSSs. Sequential algorithms are appropriate intensional refinements of Kahn-Plotkin sequential functions: it is a theorem that the quotient of the CPO of sequential algorithms by the extensional equality is isomorphic to the CPO of Kahn-Plotkin sequential functions with respect to the stable ordering.

Each approach gives rise to a cartesian closed category. We consider them in turn.

---

<sup>11</sup>Kahn and Plotkin’s goal in the pioneering paper [Kahn and Plotkin, 1978] was *not* the systematic analysis of higher-type sequential computation. Rather, they were concerned with computational issues in the setting of stream-like domains.

## 4.2 dI-domains and stable functions

Berry's stable functions [Berry, 1978a; Berry, 1979] can be thought of as occupying a place between Scott-continuous functions and Kahn-Plotkin sequential functions in terms of degree<sup>12</sup> of sequentiality. One could say that stability is an approximation to higher-type sequentiality. Intuitively, a function is stable if a *definite* amount of information is needed from the argument in order to obtain a given approximation of the result. To paraphrase Berry, from a computational point of view, stability implies the existence of "optimal computations", but not the existence of "optimal computational rules" (in the sense of an algorithm) which would correspond to sequentiality.

Berry was, at least in part, motivated by the syntactic stability property of PCF (see Section 2). Consider the following condition which is valid in the standard continuous function space model of PCF:

For any PCF-term  $M$ , whenever  $e \sqsubseteq \mathcal{A}[M]$ , then there is a least PCF-term  $M' \leq_n M$  such that  $e \sqsubseteq \mathcal{A}[M']$ .

This observation can be made precise in abstract domain-theoretic terms. Let  $D$  and  $E$  be CPOs. A continuous function  $f : D \rightarrow E$  is said to be *stable* if for every  $e \in E$  and  $d \in D$  such that  $e \sqsubseteq f(d)$ , the set

$$\{x \in D : x \sqsubseteq d, e \sqsubseteq f(x)\}$$

has a least element which we denote as  $m(e, f, d)$ . Intuitively, the function  $f$  is stable if and only if whenever  $e \sqsubseteq f(d)$ , a definite amount of information is needed from the input argument  $d$  in order to "reach"  $e$  by the application of  $f$ . This requisite amount of information is the least element of the set  $\{x \in D : x \sqsubseteq d, e \sqsubseteq f(x)\}$ . The standard example of a continuous function which is not stable is the parallel-or function  $p\text{-or}$ . To see this, just take  $e$  to be  $\top$  and  $d$  to be  $\langle \top, \top \rangle$ . The set  $\{x \sqsubseteq d : e \sqsubseteq p\text{-or}x\}$  has two minimal elements, namely  $\langle \perp, \top \rangle$  and  $\langle \top, \perp \rangle$ ; but it has no least element.

In CPOs with richer structure, stability may be expressed in more "algebraic" terms. Recall that a subset  $X$  of a CPO  $D$  is said to be consistent if  $X$  has an upper bound in  $D$ . Restricted to functions between consistently complete CPOs, stability is equivalent to the preservation of the greatest lower bounds of consistent sets.

**Lemma 4.2.1.** *Let  $D$  and  $E$  be consistently complete CPOs. A continuous function  $f : D \rightarrow E$  is stable if and only if  $f(\sqcap X) = \sqcap f(X)$  for any non-empty consistent subset  $X$  of  $D$ .* ■

Let  $D$  and  $E$  be CPOs such that any pair of compatible elements has a greatest lower bound. A continuous function  $f : D \rightarrow E$  is said to be

<sup>12</sup>Sazonov introduced a notion of degree of sequentiality in [Sazonov, 1975]. Unfortunately, the treatment therein is hard to understand. Here, we use it in a completely informal way.



conditionally multiplicative or simply CM if  $f(x \sqcap y) = f(x) \sqcap f(y)$  for every compatible pair of elements  $x$  and  $y$ .

**Lemma 4.2.2.** *Suppose further that  $D$  and  $E$  are  $\omega$ -algebraic, and that  $D$  satisfies the following:*

**Axiom (I)** *For any compact element  $d$  of  $D$ , the set  $\{x \in D : x \sqsubseteq d\}$  is finite.*

*Then a continuous function is stable if and only if it is CM.* ■

Stable functions over arbitrary CPOs do compose.

**Lemma 4.2.3.** *Let  $f : D \rightarrow D'$  and  $g : D' \rightarrow D''$  be continuous stable maps over CPOs. For any  $d \in D$  and  $d'' \in D''$ ,  $m(d'', g \circ f, d) = m(m(d'', g, f(d)), f, d)$ .* ■

Proofs of results in this subsection can be found in [Gunter, 1992, §5.2], [Girard et al., 1989, Ch. 8] (in the more specific case of coherence spaces and stable functions), or [Bucciarelli, 1993, Ch. 2].

CPOs with stable functions form a category with binary products. However, with respect to the standard extensional (or pointwise) order, stable functions between CPOs do not form a cartesian closed category. Exponentiation in the category fails because function application

$$\text{app} : [D \rightarrow_s E] \times D \rightarrow E$$

is not in general a stable map, where  $[D \rightarrow_s E]$  is the set of continuous stable maps from  $D$  to  $E$ . To see this, consider three maps  $\perp, \text{id}, \top$  from the two-element lattice  $\mathbb{O} = \{\perp, \top\}$  to itself.  $\perp$  and  $\top$  are the least and greatest functions respectively with respect to the extensional ordering, and  $\text{id}$  is the identity map. Observe that the element  $m(\top, \text{app}, \langle \top, \top \rangle)$  is undefined — the set  $\{\langle f, d \rangle : f \sqsubseteq \top, d \sqsubseteq \top, \top \sqsubseteq f(d)\}$  has two minimal elements, namely  $\langle \top, \perp \rangle$  and  $\langle \text{id}, \top \rangle$ . With respect to the extensional ordering, application is also not CM.

**dI-domains** To rectify the situation, we modify the ordering between stable functions. First, we introduce dI-domains. A dI-domain  $D$  is a Scott domain (= consistently complete  $\omega$ -algebraic CPO) satisfying axioms (d) and (I) as follows (hence, the name dI-domain):



**Axiom (d)** For any  $d, e, f \in D$ , whenever  $e \uparrow f$ , then  $d \sqcap (e \sqcup f) = (d \sqcap e) \sqcup (d \sqcap f)$ .

**Axiom (I)** Every compact element dominates finitely many elements, i.e. for any finite element  $d$  of  $D$ , the set  $\{x \in D : x \sqsubseteq d\}$  is finite.

Note that axiom (I) is very often referred to as (F). The dI-domains have an equivalent characterization in terms of prime algebraicity. Recall that an element of a CPO  $D$  is a *prime* if for any subset  $X$  of  $D$  such that  $\bigsqcup X$  exists and  $x \sqsubseteq \bigsqcup X$ , then  $x \sqsubseteq y$  for some  $y \in X$ . A CPO  $D$  is said to be *prime algebraic* if every element  $x$  is the lub of the set of prime elements dominated by  $x$ .

**Lemma 4.2.4.** *A Scott domain which satisfies the axiom (I) is prime algebraic if and only if it also satisfies the axiom (d).* ■

This lemma (the implication “ $\Leftarrow$ ”) is due to Winskel [Winskel, 1987]. It may be proved by showing that in a dI-domain, primes are those compact elements which cover exactly one element. Given dI-domains  $D$  and  $E$ , the *stable order*  $\sqsubseteq_s$  over  $[D \rightarrow_s E]$  is defined as follows: for stable functions  $f$  and  $g$ ,

$$f \sqsubseteq_s g \stackrel{\text{def}}{=} \begin{cases} \forall d \in D. f(d) \sqsubseteq g(d) & \& \\ \forall d, e \in D. d \sqsubseteq e \implies f(d) = f(e) \sqcap g(d). \end{cases}$$

Stable ordering is not an arbitrary invention, nor is dI-domain an accidental structure. The stable ordering is in fact the largest ordering included in the extensional ordering which makes the standard evaluation map stable. Further, we have:

**Theorem 4.2.5 (Berry).** *The category of dI-domains and stable functions is cartesian closed.* ■

The category of dI-domains and stable maps may be characterized in a very satisfactory way: it is the largest subcategory of Scott domains with stable maps as morphisms which is cartesian closed (see [Berry, 1978a]).

A stable function is completely determined by its *trace*.<sup>13</sup> First, some notation. For any dI-domain  $D$ , we write the collection of its compact elements as  $K(D)$  and the collection of its prime elements as  $|D|$ . The *trace* of a stable function  $f : D \rightarrow E$  is defined as follows:

<sup>13</sup>Though the idea of the trace of a stable function was implicit in the earlier work of Berry and Winskel, the name “trace” was introduced by Girard in [Girard, 1986] where he studied stable functions between qualitative domains as a model of *System F*.

$$tr(D) \stackrel{\text{def}}{=} \{ (d, e) : d \in K(D), e \in |E|, d \text{ is minimal such that } e \sqsubseteq f(d) \}.$$

The trace of a stable function characterizes the function in the following way:

**Proposition 4.2.6.** *Let  $f, g : D \rightarrow E$  be stable functions between dI-domains.*

- (i)  $f(x) = \bigsqcup \{ e : (d, e) \in tr(f) \text{ for some } d \sqsubseteq x \}.$
- (ii)  $f \sqsubseteq_s g$  if and only if  $tr(f) \subseteq tr(g).$

■

Does the stable function space give rise to a fully abstract model of PCF? Consider the first-order function  $\text{gustave} : \mathbb{B}_\perp \times \mathbb{B}_\perp \times \mathbb{B}_\perp \rightarrow \mathbb{B}_\perp$  which is defined as the least monotonic function satisfying

$$\begin{aligned} \text{gustave}(\perp, t, f) &= t, \\ \text{gustave}(f, \perp, t) &= t, \\ \text{gustave}(t, f, \perp) &= t. \end{aligned}$$

Note that the three 3-tuples are mutually incompatible. From any one of the three 3-tuples, the other two may be obtained by cyclically permuting the components. The function  $\text{gustave}$  is stable but not Kain-Plotkin sequential. Hence the stable function space model (in which functions are ordered stably) cannot be fully abstract for PCF.

Is there any extension of the language PCF for which the stable function space model is fully abstract? This is a natural question to ask in view of Plotkin's result that the continuous function space model is not fully abstract for PCF, but it is for PCF extended by a parallel conditional. This problem has been extensively studied by Jim and Meyer [Jim and Meyer, 1991]. Their main result in that paper provides a negative answer to the above question. They show that every stable function space model which is adequate for a conservative extension of PCF defined by "PCF-like rewrite rules" is not (equationally) fully abstract.

The idea of stability has been reinvented by Girard in the mid-1980's to provide models for *System F* or second-order polymorphic  $\lambda$ -calculus [Girard, 1986]. The motivation in this case could be traced back to Girard's earlier work on dilators [Girard, 1981]. Girard has also introduced a number of subcategories of the category of dI-domains and stable functions which are cartesian closed. For example, the category of qualitative domains [Girard, 1986] and the category of coherence spaces (see [Girard *et al.*, 1989]); both categories have stable maps as morphisms. The latter led Girard to the invention of linear logic [Girard, 1987]. For a treatment of stability of a categorical flavour, see *e.g.* [Taylor, 1990].

### 4.3 Sequential algorithms

Kahn and Plotkin's concrete data structure provides an innovative conceptual framework for a new notion of sequentiality. While Kahn-Plotkin sequentiality is in principle applicable to functions at higher types, the category of DCDSS (deterministic concrete data structures) and Kahn-Plotkin sequential functions is not cartesian closed. Is there a cartesian closed category of DCDSS?

The answer to this question is found in Berry and Curiens' sequential algorithms [Berry and Curien, 1982]. Given two DCDSS  $M$  and  $M'$ , they define a new DCDSS  $M \Rightarrow M'$  whose states are the sequential algorithms from  $M$  to  $M'$ . A sequential algorithm may be characterized in terms of a function describing the associated input-output behaviour together with a computation strategy. This characterization enables the definition of a categorical composition of sequential algorithms which is technically the most demanding result in Berry and Curien's work on sequential algorithms.

The relationship between sequential algorithms and Kahn-Plotkin sequential functions may be stated exactly: the CPO of sequential functions ordered by the stable ordering is isomorphic to the CPO of sequential algorithms quotiented by extensional equality. Therefore, it is right to think of a sequential algorithm as an intensional refinement of a Kahn-Plotkin sequential function. Proofs of the results in this subsection can be found in [Berry and Curien, 1982] and [Curien, 1993a].

Let  $M = \langle C, V, E, \vdash \rangle$  and  $M' = \langle C', V', E', \vdash' \rangle$  be two DCDSS. The structure  $M \Rightarrow M' = \langle C'', V'', E'', \vdash'' \rangle$  is defined as follows:

- $C''$  is the set of pairs, written  $xc'$ , where  $x$  is a finite state of  $M$ , and  $c'$  is a cell of  $M'$ ;
- $V''$  is the disjoint union of  $C$  and  $V'$ ; the two kinds of elements are represented as *valof*  $c$  and *output*  $v'$  respectively;
- $E''$  is defined by the following bi-implications:

$$(xc', \text{valof } c) \in E'' \iff c \in \mathbf{A}(x) \quad \text{"valof events",}$$

$$(xc', \text{output } v') \in E'' \iff (c', v') \in E' \quad \text{"output events";}$$

- the enabling  $\vdash''$  is of two types defined as follows:
  - \* type "valof":  $(xc', \text{valof } c) \vdash'' yc'$  if and only if  $x \prec_c y$ ;
  - \* type "output":  $(x_1c'_1, \text{output } v'_1), \dots, (x_nc'_n, \text{output } v'_n) \vdash'' xc'$  if and only if

$$x = \bigcup_{1 \leq i \leq n} x_i \quad \text{and} \quad (c'_1, v'_1), \dots, (c'_n, v'_n) \vdash' c'.$$

If  $M$  and  $M'$  are filiform DCDSS, then the definition may be simplified. We tabulate the data as follows:

	$M$	$M'$	$M \Rightarrow M'$
Cells	$c$	$c'$	$xc'$ for $x \in D(M)^{\text{fin}}$
Values	$v$	$v'$	valof $c$ , output $v'$
Events	$(c, v)$	$(c', v')$	$(xc', \text{valof } c)$ for $c \in \mathbf{A}(x)$ $(xc', \text{output } v')$ for $(c', v') \in E'$ ,
Enabling		$(c'_1, v'_1) \vdash c'$	$(xc', \text{valof } c) \vdash' yc'$ if $x \prec_c y$ $(xc'_1, \text{output } v'_1) \vdash'' xc'$

We can think of an event of  $M \Rightarrow M'$  of the shape  $(xc', ?)$  where  $?$  is either “valof  $c$ ” or “output  $v'$ ” as an atomic computation which receives input  $x$  from  $M$  and whose ultimate goal is to produce something to fill the cell  $c'$ . Exactly what the atomic computation does depends on what  $?$  is. Intuitively, if  $?$  is “valof  $c$ ”, then more information is needed from the input  $x$  (and specifically, the value of the cell  $c$  in the state  $x$ ) before output can be produced in  $c'$ . If  $?$  is “output  $v'$ ”, then the value  $v'$  is offered immediately so as to fill the cell  $c'$ .

A state of  $M \Rightarrow M'$  is called a *sequential algorithm* from  $M$  to  $M'$ . There is an “input-output function” associated with each sequential algorithm. For any state  $x$  of  $M$  and any state  $a$  of  $M \Rightarrow M'$ ,

$$a \cdot x \stackrel{\text{def}}{=} \{ (c', v') : \exists y \subseteq x. (yc', \text{output } v') \in a \}.$$

We say that  $a \cdot x$  is the result of applying  $a$  to  $x$ . It is worth pausing to ponder over the definition of a sequential algorithm. Formally, it is a collection of pairs (events) which have one of two shapes:  $(xc', \text{valof } c)$  or  $(xc', \text{output } v')$ , i.e. valof event or output event. But now, consider the effect of applying a sequential algorithm  $a$  to the “argument”  $x$ , as in the above definition. Note that the valof events do not play a role in the definition of  $a \cdot x$ . This is not surprising since the definition is intended to capture the extensional behaviour of the algorithm  $a$ ; the output events are “intensional details” which rightly get suppressed.

**Proposition 4.3.1.** *Using the same notations as before, suppose  $M$  is a CDS and  $M'$  a DCDS; then  $a \cdot x$  is a state of  $M'$  and the function  $x \mapsto a \cdot x$  is a continuous and stable function from  $D(M)$  to  $D(M')$ . Also,  $M \Rightarrow M'$  as defined is a DCDS.* ■

If  $M'$  is not stable, then there is no guarantee that the image of the input-output function  $x \mapsto a \cdot x$  is always a state, see [Curien, 1993a, p. 217]. For this reason, we shall henceforth only consider sequential algorithms between DCDSS.

For example, the formal descriptions of the two addition algorithms,



“left-add” and “right-add”, are as follows:

$$\begin{aligned}
 \text{l-add} &= \{ (\{ \}?, \text{valof } ?\cdot 1) \} \\
 &\cup \{ (\{ (?\cdot 1, i) \}?, \text{valof } ?\cdot 2) : i \in \omega \} \\
 &\cup \{ (\{ (?\cdot 1, i), (?\cdot 2, j) \}?, \text{output } i + j) : i, j \in \omega \}; \\
 \text{r-add} &= \{ (\{ \}?, \text{valof } ?\cdot 2) \} \\
 &\cup \{ (\{ (?\cdot 2, j) \}?, \text{valof } ?\cdot 1) : j \in \omega \} \\
 &\cup \{ (\{ (?\cdot 2, j), (?\cdot 1, i) \}?, \text{output } i + j) : i, j \in \omega \}.
 \end{aligned}$$

As an illustration, we give a reading of the “left-add” algorithm. At the start of the computation, corresponding to the event  $(\{ \}?, \text{valof } ?\cdot 1)$ , the algorithm asks for the value of the left argument which corresponds to the cell  $?\cdot 1$ . Suppose the left argument supplies a value, say  $i$ . The  $\text{valof}$  event  $(\{ (?\cdot 1, i) \}?, \text{valof } ?\cdot 2)$  then indicates that the algorithm now asks for the value of the right argument which corresponds to the cell  $?\cdot 2$ . Suppose the right argument supplies a value  $j$ . The output event  $(\{ (?\cdot 1, i), (?\cdot 2, j) \}?, \text{output } i + j)$  of the algorithm then concludes the computation by returning the value  $i + j$ .

**Theorem 4.3.2 (Berry-Curien).** *The category of DCDSs (deterministic concrete data structures) and sequential algorithms is cartesian closed. It has as a full subcategory the cartesian closed category of filiform DCDSs. ■*

The category of DCDSs and sequential algorithms does not yield an extensional model of PCF. We have already seen that “left-add” and “right-add” are two distinct sequential algorithms, even though they have the same extensional behaviour as represented by the standard addition function. Nonetheless, the tantalizing question is this: can this category be turned into an order-extensional, continuous, fully abstract model for PCF? Curien set out to answer just this question. We chart his progress briefly; see [Curien, 1993a, §4.3–4.4] for more details.

- First, a cartesian category **BISEQ**<sub>0</sub> whose objects are a kind of DCDS equipped with two orderings following the idea of Berry’s “bi-structure” [Berry, 1978b] was constructed. The arrows are extensionally ordered functions.
- In order to get *complete* partially ordered domains, a full subcategory of **BISEQ**<sub>0</sub> called **BISEQ**<sub>1</sub> was then defined which turned out to be cartesian closed. This yields a continuous and order-extensional model for PCF.

The only missing step now to full abstraction is definability of compact elements. Unfortunately, the model associated with **BISEQ**<sub>1</sub> falls short of full abstraction: there are compact elements of the model which are not



PCF-definable. These are the ingenious counter-examples  $A^1$ ,  $A^2$  and  $A^3$  of Curien. To get a flavour, we will only briefly mention algorithm  $A^1$  which is of type  $(o \Rightarrow (o \Rightarrow o)) \Rightarrow o$ ; it sends the “left-strict or” to “true” and the “right-strict or” to “false”. For a more precise definition, we refer the reader to [Curien, 1993a, pp. 357–351].

What is the programming language for which the model associated with the cartesian closed category of filiform DCDSS and sequential algorithms gives a fully abstract semantics? Berry and Curien introduced a rather unconventional functional language called **CDS** which is described in full in [Berry and Curien, 1985]. A distinctive feature of this language is that its computational observables are not restricted just to data of ground type. In fact, the call-by-need parameter-passing mechanism allows the programmer to observe as much information about *any* program fragment as he wishes. A crucial consequence of this liberal approach to performing computational observation is that it becomes possible for the programmer to ascertain the order of evaluation of input arguments. For example, though the intensionally different “right-and” and “left-and” have the same extensional behaviour as functions, they are distinguishable by an appropriate higher-type procedure definable in **CDS**. The main result of this work is that the model based on sequential algorithms is fully abstract for **CDS**.

#### 4.4 Observable algorithms and PCF with error values

Recently, drawing on their intuitions as programmers, Cartwright and Felleisen [Cartwright and Felleisen, 1992] invented the *observably sequential functions* to construct a fully abstract denotational semantics for a sequential extension of PCF which they call SPCF. Curien [Curien, 1992] immediately realized that the observably sequential functions were a natural extensional refinement of sequential algorithms. This is remarkable because the sequential algorithms (defined exactly as before) being considered in the extended setting, which are called *observable algorithms*, are still very much intensional in nature, and are most succinctly represented as a kind of decision tree. The key to this surprising development is the presence of “error values” in the semantic domains. To ensure a well-behaved mechanism of function application, observable algorithms are required to “percolate errors to the top” when they are applied to arguments. A main result is that the category of DCDSS (with error values) and observable algorithms is cartesian closed. The associated model is not fully abstract for PCF, but it is for an extension of PCF which has error values and escape-handling control facilities much resembling the *catch* facility in some versions of Lisp.

**Observable algorithms** We assume that a non-empty set *Err* of *error values*, or simply *errors*, has been fixed. In particular, we assume that the set of errors is disjoint from the values of all CDSS that are being considered.

Given a DCDS  $M = \langle C, V, E, \vdash \rangle$ , an *observable state* of  $M$  is a set  $x$  of pairs  $(c, w)$  where either  $(c, w) \in E$  or  $w \in \text{Err}$ , and satisfying the two conditions, namely consistency and safety, that define a state of a CDS. Note that a state is, *a fortiori*, an observable state.

Enablings are not allowed to contain errors: it is important to note that the enabling relation is part of the structure of a CDS which remains the same as before. Hence, errors occur only “at the leaves” of the tree representing the enabling relation. Given DCDSS  $M$  and  $M'$ , the observable states of  $M \Rightarrow M'$  are called *observable algorithms*. Note that the category that we are now considering has the same objects and the same product and exponentiation constructions on objects as the category of DCDSS though the maps are now different. They are the observable states of objects which are obtained by the exponentiation construction.

In order to arrive at a cartesian closed category, it is essential to have a function application which handles errors properly. So the application of sequential algorithms will not do. For every observable algorithm  $a$  from DCDSS  $M$  to  $M'$ , we define an *observable input-output function* from  $D_{\text{obs}}(M)$  to  $D_{\text{obs}}(M')$  in the following way:

$$\begin{aligned} a \cdot x &\stackrel{\text{def}}{=} \{ (c', v') : \exists y \subseteq x. (yc', \text{output } v') \in a \} \\ &\cup \{ (c', e) : \exists y \subseteq x. (yc', e) \in a \} \\ &\cup \{ (c', e) : \exists y \subseteq x. (yc', \text{valof } c) \in a, (c, e) \in x \}. \end{aligned}$$

The first clause of the definition is exactly the same as that of the input-output function associated with a sequential algorithm which we have just seen in the previous subsection. The second and third clauses tell us how errors in the “function”  $a$  and “argument”  $x$  respectively are to be propagated upwards. This ensures that the observable input-output function is well-defined, *i.e.* it maps an observable state to another; it is also continuous.

A function between the observable states of two DCDSS  $M$  and  $M'$  is said to be *observably sequential* if it satisfies the following properties:

- *continuity*. If  $c'$  is filled in  $f(x)$  then  $c'$  is filled in  $f(y)$  for some finite  $y \subseteq x$ .
- *sequentiality*. If  $c'$  is accessible from  $f(x)$  and that  $c'$  is filled in  $f(z)$  for some  $x \subseteq z$ , then there exists some cell  $c$  accessible from  $x$  such that the following holds: whenever  $c'$  is filled in  $f(y)$ , then  $c$  is filled in  $y$ , for any  $y \supseteq x$ ; we call  $c$  a *sequentiality index* for  $x$  at  $c'$ .
- *error propagation*. If  $c$  is a sequentiality index for  $x$  at  $c'$ , then  $(c', e) \in x \cup \{(c, e)\}$ , for every error  $e \in \text{Err}$ .

The next result is noteworthy since it greatly simplifies the definition of the categorical composition of observable algorithms compared to that of

sequential algorithms.

**Proposition 4.4.1.** *Every observably sequential function is the observable input-output function of a (unique) observable algorithm.* ■

Given the preceding proposition, it is a small step to prove the next.

**Theorem 4.4.2 (Curien).** *The category of DCDSs and observable algorithms is a cartesian closed category which has as a full subcategory the cartesian closed category of filiform DCDS. Both categories are extensional. If the set **Err** of errors has at least two elements, then the categories are order-extensional.* ■

Proof of the theorem can be found in [Cartwright *et al.*, 1994].

Cartwright and Felleisen extended PCF with the following additional constructs:

$$M ::= \dots \mid \text{error} \mid \text{catch}(M).$$

The typing rules are extended by

$$\text{error} : \iota \qquad \frac{M : \sigma}{\text{catch}(M) : \iota}.$$

The definition of evaluation context is extended by an additional clause as follows:  $m \geq 0$

$$E ::= \dots \mid \text{catch}(\lambda x_1 \dots x_m.E).$$

There are three more reduction rules: for  $m \geq 0$ ,

$$\begin{aligned} E[\text{error}] &\rightarrow \text{error}, \\ \text{catch}(\lambda x_1 \dots x_m.n) &\rightarrow m + n, \\ \text{catch}(\lambda x_1 \dots x_m.E[x_i]) &\rightarrow i - 1, \qquad i \leq m. \end{aligned}$$

The first reduction rule makes precise the behaviour of an evaluation context whenever its hole is filled by an error value: the error is immediately “percolated up” to the top as the value of the expression. Suppose  $F$  is a closed term of type  $(\sigma_1, \dots, \sigma_m, \iota)$ , and  $M_i$  is an arbitrary closed term of type  $\sigma_i$ , for each  $1 \leq i \leq m$ . Either  $F$  reduces to  $\lambda x_1 \dots x_m.k$ , in which case the value of  $\text{catch}(F)$  is  $m + k$ ; or the evaluation of  $FM_1 \dots M_m$  begins with the evaluation of  $M_i$ , for some  $i$ . Then the value of  $\text{catch}(F)$  is  $i - 1$ . So  $\text{catch}(-)$  is a device which makes the order of evaluation ascertainable (and expressible) within the language. For example, the values of  $\text{catch}(\text{l-add})$  and  $\text{catch}(\text{r-add})$  are 0 and 1 respectively.

The main result of this work is the following full abstraction result.

**Theorem 4.4.3 (Cartwright-Curien-Felleisen).** *The model of observable algorithms is fully abstract for PCF extended by errors.* ■

**Remark** Curien *et al* also showed that the original model of sequential algorithms is fully abstract for PCF extended by catch only. Though the concept of error values is instrumental to the proof of full abstraction, sequential algorithms are definable without using errors.

## 4.5 Towards a characterization of sequentiality

Kahn-Plotkin sequentiality and Berry and Curien's sequential algorithm are both formulated in the framework of the rather concrete setting of CDS. Two questions may be raised:

- (Q1) CDS is quite a complicated structure but its components, namely cells, values and an enabling relation, are essential to the formulation of two seminal ideas: Kahn-Plotkin sequentiality and sequential algorithm. It would be desirable if the ideas underpinning CDS could be made more abstract. Can the "intensional" features of a CDS be expressed equivalently in terms of a set of extrinsic properties defined on a class of CPOs (say, the dI-domains)?
- (Q2) Can the definition of the Kahn-Plotkin sequential function be characterized in "algebraic" terms, or in terms of some preservation property? Even better, is there a topology whose associated notion of continuity is equivalent to Kahn-Plotkin sequentiality?

In a series of papers, Bucciarelli and Ehrhard, [1993a; 1993b; 1993], set out to answer these questions systematically. Their first key insight may be stated simply:

"cells = (a kind of) linear maps".

They replace the notion of cell by a kind of linear map on consistently complete domains (as we will see later, other conditions force these domains to be dI-domains). The upshot is a more abstract and more general notion of cell or place, as we called it earlier. As an illustration, consider the CDS representation of the flat CPO of natural numbers. There is only one cell,  $*$  say, and the values are  $\{0, 1, 2, \dots\}$ . A number  $n$  is then represented as the cell  $*$  which is filled with the value  $n$ . In this simple example, Bucciarelli and Ehrhard's proposal is to replace the cell  $*$  by the linear map (a linear map is a stable map which preserves the lub of any finite, compatible subset) from the flat CPO of natural numbers to the Sierpinski domain  $\mathbf{O}$  (i.e. the two-point dI-domain  $\perp \sqsubseteq \top$ ) whose trace is

$$\{(0, \top), (1, \top), (2, \top), \dots\}.$$

**Sequential structures** Sequential structure is Bucciarelli and Ehrhard's answer to the first question (Q1). This structure may be understood as



an abstract device which captures the notions of cells (= questions) and values (= answers) of a CDS. We give the definition of sequential structure in stages. First, a *linear map* between two dI-domains  $D$  and  $E$  is a stable function such that

- $f(\perp) = \perp$ , and
- $f(x \sqcup y) = f(x) \sqcup f(y)$  for any pair of compatible elements  $x$  and  $y$ .

A *sequential pre-structure* is a pair  $X = (X_*, X^*)$  where  $X_*$  is a consistently complete CPO and  $X^*$  is a set of linear functions from  $X_*$  to  $\mathbf{O}$  satisfying the following properties:

- The constantly bottom function is in  $X^*$ .
- $X^*$  *separates*  $X_*$ : whenever  $x$  and  $y$  in  $X_*$  are compatible, then

$$x \sqsubseteq y \quad \Longleftrightarrow \quad \alpha(x) \sqsubseteq \alpha(y), \text{ for every } \alpha \in X^*.$$

Given a sequential pre-structure  $X$ , an element  $x$  of  $X_*$  is said to be *finite* if the set

$$\{\alpha \in X^* : \alpha(x) = \top\} \text{ is finite.}$$

A finite element of  $X$  in this sense is to be distinguished from a compact element of  $X_*$ . Note that finite element is a notion defined relative to  $X_*$ . Finite elements enjoy a nice property reminiscent of the axiom (I) which is not in general valid for compact elements.

**Lemma 4.5.1.** *If  $c \in X_*$  is finite, then so is any  $x \sqsubseteq c$ , and  $c$  has only a finite number of approximants. Further,  $c$  is compact.* ■

We direct the reader to [Bucciarelli and Ehrhard, 1993b] for proofs of results mentioned in this subsection. It is easy to show that in a sequential pre-structure  $X$ , for any element  $x$  of  $X_*$ , the collection of finite approximants of  $x$  is directed. We can now define sequential structure. A pre-sequential structure  $\langle X_*, X^* \rangle$  is called a *sequential structure* if every element  $x$  of  $X_*$  is the lub of its finite (in the above specialized sense)  $\sqsubseteq$ -approximants; that is to say,  $x = \bigsqcup \{c : c \sqsubseteq x, c \text{ is finite}\}$ .

**Proposition 4.5.2.** *If  $X$  is a sequential structure, then  $X_*$  is a dI-domain.* ■

How well does the abstract notion of sequential structure generalize the concrete framework of CDS? Sequential structure turns out to be a very general device for expressing sequential computation. To begin with, we can express a CDS quite satisfactorily as a sequential structure. Given a DCDS  $M = \langle C, V, E, \vdash \rangle$ , the space of data or answers is  $D(M)$ , the collection of states of the DCDS  $M$ , and questions are represented by cells: a cell  $c$  now stands for the function that maps a state  $x$  to  $\top$  if and only if  $c$  is filled in  $x$ . It can be shown that  $\langle D(M), C \cup \{\perp\} \rangle$  defines a sequential



structure. Another example is Girard's coherence spaces, see [Bucciarelli and Ehrhard, 1993b].

Is there a way of expressing Kahn-Plotkin sequentiality in the setting of sequential structures? Consider the notion of CDS. Think of an element  $\alpha$  of  $X^*$  as a question, and an element  $x$  of  $X_*$  as an answer. So  $x$  is an appropriate answer of the question  $\alpha$  precisely when  $\alpha(x) = \top$ . It is also helpful to think of a linear function  $\alpha \in X^*$  as a property of elements of  $X_*$ . Hence we have

$$x \in \alpha \quad \text{means} \quad \alpha(x) = \top \quad \text{"}x \text{ answers the question } \alpha\text{"}.$$

And of course,  $x \notin \alpha$  means  $\alpha(x) = \perp$ . Even though the following definition is set in the framework of sequential structures, it really just mimics the definition of the Kahn-Plotkin sequential function between CDSS.

Let  $(X, X^*)$  and  $(Y, Y^*)$  be sequential structures. A continuous function  $f : X \rightarrow Y$  is *sequential* if for any  $x \in X$  and for any  $\beta \in Y^*$ , whenever  $f(x) \notin \beta$ , there is some  $\alpha \in X^*$  such that

- $x \notin \alpha$  and
- for any  $x' \sqsupseteq x$ , whenever  $f(x') \in \beta$ , then  $x' \in \alpha$ .

Following the terminology of Kahn-Plotkin sequentiality, we call the question  $\alpha$  a *sequentiality index* of  $f$  for  $x$  at  $\beta$ . Using our answer-versus-question reading of  $(X, X^*)$ , we paraphrase the definition as follows: if  $f(x)$  does not answer the given question  $\beta$ , then there is a question  $\alpha$  not answered by  $x$  such that it *must* be answered by any  $x' \sqsupseteq x$  whenever  $f(x')$  answers  $\beta$ .

In fact, even sequential algorithms may be defined in the setting of sequential structures extended with an abstract notion of enabling (defined axiomatically). We refer the reader to [Bucciarelli and Ehrhard, 1993b] for the definition. The high point of the theory of sequential structures is this:

**Theorem 4.5.3.** *The category of sequential structures with enabling and sequential algorithms is cartesian closed.* ■

Thus, the goal of generalizing the intuitions of Kahn-Plotkin sequentiality and sequential algorithms to a setting wider and more abstract than CDS is achieved.

**Linear coherence** Now, we turn to the second question (Q2). Let  $(X, X^*)$  be a sequential structure. A finite subset  $A$  of  $X$  is said to be *linearly coherent* provided for any  $\alpha \in X^*$ ,  $\sqcap A \in \alpha$  whenever  $A \subseteq \alpha$ . (The last expression is a shorthand for  $a \in \alpha$  for every  $a \in A$ .) A continuous function  $f : X \rightarrow Y$  is said to be *linearly stable* if for any linearly coherent subset  $A$  of  $X$ ,  $f(A)$  is a linearly coherent subset of  $Y$  and  $f(\sqcap A) = \sqcap f(A)$ . The sequential function between sequential structures may be characterized

algebraically in terms of linear stability.

**Proposition 4.5.4.** *A function between sequential structures is sequential if and only if it is linearly stable.* ■

**dI-domains and coherence** Let  $D$  be a dI-domain. A collection  $C$  of finite non-empty subsets of  $D$  is called a *coherence* whenever the following conditions are satisfied:

- for every element  $x$  of  $D$ , the singleton set  $\{x\}$  is in  $C$ ;
- for any  $X \in C$  and for any finite subset  $A$  of  $D$ , whenever

$$\forall a \in A. \exists x \in X. a \sqsubseteq x \&$$

$$\forall x \in X. \exists a \in A. a \sqsubseteq x$$

then  $A$  is in  $C$ ;

- for any directed subsets  $A_1, \dots, A_n$  of  $D$  such that  $\{a_1, \dots, a_n\}$  is in  $C$  whenever  $a_1 \in A_1, \dots, a_n \in A_n$ , then  $\{\sqcup A_1, \dots, \sqcup A_n\}$  is in  $C$ .

Whenever the coherence of a dI-domain is understood from the context, we write it as  $C(D)$ .

**Strongly stable functions** Let  $D$  and  $E$  be dI-domains with coherence. A continuous function  $f$  from  $D$  to  $E$  is said to be *strongly stable* if for any  $A \in C(D)$ ,  $f(A) \in C(E)$  and  $\sqcap f(A) = f(\sqcap A)$ . In the general setting of dI-domains with coherence, the following very satisfactory result on strong stability may be obtained.

**Theorem 4.5.5.** *The category whose objects are dI-domains with coherence and whose morphisms are strongly stable functions is cartesian closed.* ■

This category gives rise to a model of PCF. In [Ehrhard, 1994b], Ehrhard shows that any strongly stable function which arises from the model is the “extensional component” of a sequential algorithm. More precisely, a cartesian closed category is constructed whose objects are triples  $\langle E, X, \pi \rangle$ . In such a triple,  $E$  is a sequential structure,  $X$  is a hypercoherence, and  $\pi$  is a function from  $E_*$  (the space of points of  $E$ ) to  $\text{qD}(X)$ , the qualitative domain induced by  $X$ . The function  $\pi$  is required to be linear, strongly stable (with respect to both the linear coherence induced by  $E^*$  on  $E_*$ , as well as the coherence induced by the hypercoherence  $X$  on  $\text{qD}(X)$ ) and onto. (Hypercoherence (see [Ehrhard, 1994a]) is a simplified framework for dealing with strong stability which also gives rise to a model of linear logic. A hypercoherence is a hypergraph which naturally gives rise to a qualitative domain equipped with a coherence.) The intuition is this:  $E_*$  is the space of sequential algorithms,  $\text{qD}(X)$  is a space of strongly stable functions, and  $\pi$  is the “forgetful” operation which sends any sequential algorithm onto its generalized extensional component. In this set up, the

force of the function  $\pi$  being onto is that any strongly stable function is in some sense the extensional component of a sequential algorithm.

**Topological approaches** Brookes and Geva [Brookes and Geva, 1992b] have adopted a topological approach in an attempt to characterize sequentiality. They propose a general definition of sequential functions on Scott domains, characterized by a generalized notion of topology (along the lines of [Lamarche, 1993]), based on sequential open sets. This notion of sequential function turns out to coincide with the Kahn–Plotkin notion of sequential function when restricted to distributive concrete domains, and considerably expands the class of domains for which sequential functions may be defined. Ordered stably, the sequential functions between two dI-domains form a dI-domain (the analogous property fails for Kahn–Plotkin sequential functions). However, the category of dI-domains and sequential functions is not cartesian closed because application is not sequential. They have another related line of research which consists in the development of a theory of intensional semantics. In [Brookes and Geva, 1992a], they generalize Kahn and Plotkin’s concrete data structure to obtain a cartesian closed category of generalized concrete data structure and continuous functions. Using this cartesian closed category as an extensional framework, they define an intensional framework — a cartesian closed category of generalized concrete data structure and parallel algorithms which may be seen as a generalization of sequential algorithms. This construction encapsulates a notion of intensional behaviour as a computational comonad.

## Acknowledgements

I would like to thank the following colleagues who provided valuable criticisms: Samson Abramsky, Richard Bird Antonio Bucciarelli, Pierre-Louis Curien,

Thomas Ehrhard, Shai Geva, Martin Hyland, Radha Jagadeesan, François Lamarche, Pasquale Malacaria, Andrew Moran, Andrew Pitts, Gordon Plotkin, Eike Ritter and Allen Stoughton. I am especially indebted to Martin Hyland for his intellectual companionship over the past couple of years; and to Pierre-Louis Curien and Andrew Moran for reading a preliminary draft of this chapter with far more care than it deserved.

This work was carried out when I was a research fellow at the Computer Laboratory, University of Cambridge, while on leave from the National University of Singapore. This project was supported in part by grants from the Science and Engineering Research Council (UK), Esprit Basic Research Action 6811 “Categorical Logic in Computer Science II”, and a Prize Fellowship from Trinity College, Cambridge.

## References

- [Abramsky, 1990] S. Abramsky. The lazy lambda calculus. In David Turner, editor, *Research Topics in Functional Programming*. Addison-Wesley, 1990.
- [Abramsky and Ong, 1993] S. Abramsky and C.-H. L. Ong. Full abstraction in the lazy lambda calculus. *Information and Computation*, 105:159–267, 1993.
- [Barendregt, 1984] H. Barendregt. *The Lambda Calculus*. North-Holland, revised edition, 1984.
- [Berry, 1978a] G. Berry. Séquentialité de l'évaluation formelle des  $\lambda$ -expressions. In *Proc. 3ème Colloque International sur la Programmation, Paris*, 1978.
- [Berry, 1978b] G. Berry. Stable models of typed lambda calculus. In *Proc. 5th Colloquium on Algorithms, Languages and Programming*, pages 72–89. LNCS Vol. 62, Springer-Verlag, 1978.
- [Berry, 1979] G. Berry. *Modèles complètement adéquats et stables des lambda calculs typés*. PhD thesis, Université Paris VII, 1979.
- [Berry and Curien, 1982] G. Berry and P.-L. Curien. Sequential algorithms on concrete data structures. *Theoretical Computer Science*, 20:265–321, 1982.
- [Berry and Curien, 1985] G. Berry and P.-L. Curien. Theory and practice of sequential algorithms: the kernel of the programming language *cds*. In M. Nivat and J. Reynolds, editors, *Algebraic Semantics*. Cambridge University Press, 1985.
- [Berry et al., 1986] G. Berry, P.-L. Curien, and J.-J. Lévy. Full abstraction for sequential languages: the state of the art. In M. Nivat and J. Reynolds, editors, *Algebraic Semantics*, pages 89–132. Cambridge University Press, 1986.
- [Böhm, 1968] C. Böhm. Alcune proprietà delle forme  $\beta\eta$ -normali nel  $\lambda k$ -calculus. Technical Report 696, Istituto per le Applicazioni del Calcolo, Roma, 1968.
- [Brookes and Geva, 1992a] S. Brookes and S. Geva. Continuous functions and parallel algorithms on concrete data structures. In *Proc. 7th International Conference on Mathematical Foundations of Programming Semantics*, pages 326–349. LNCS Vol. 598, Springer-Verlag, 1992.
- [Brookes and Geva, 1992b] S. Brookes and S. Geva. Stable and sequential functions on Scott domains. Technical Report CMU-CS-92-121, School of Computer Science, Carnegie Mellon University, 1992.
- [Bucciarelli, 1993] A. Bucciarelli. *Sequential Models of PCF: some contributions to the domain-theoretic approach to full abstraction*. PhD thesis, Dipartimento di Informatica, Università di Pisa, 1993.



- [Bucciarelli and Ehrhard, 1991] A. Bucciarelli and T. Ehrhard. Sequentiality and strong stability. In *Proc. 6th Annual IEEE Symp. Logic in Computer Science*, pages 138–145. IEEE Computer Society Press, 1991.
- [Bucciarelli and Ehrhard, 1993a] A. Bucciarelli and T. Ehrhard. Sequentiality in an extensional framework. Unpublished manuscript, 1993.
- [Bucciarelli and Ehrhard, 1993b] A. Bucciarelli and T. Ehrhard. A theory of sequentiality. *Theoretical Computer Science*, 113:273–292, 1993.
- [Cartwright and Felleisen, 1992] R. Cartwright and M. Felleisen. Observable sequentiality and full abstraction (preliminary version). In *Proc. 19th ACM Symp. Principles of Programming Languages*, pages 328–342. ACM Press, 1992.
- [Cartwright et al., 1994] R. Cartwright, P.-L. Curien, and M. Felleisen. Fully abstract semantics for observably sequential languages. *Information and Computation*, 111:297–401, 1994.
- [Church, 1940] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.
- [Curien, 1992] P.-L. Curien. Sequentiality and full abstraction. In M. P. Fourman et al., editor, *Applications of Categories in Computer Science*, pages 66–94. Cambridge University Press, 1992.
- [Curien, 1993a] P.-L. Curien. *Categorical Combinators, Sequential Algorithms, and Functional Programming*. Progress in Theoretical Computer Science Series. Birkhäuser, second edition, 1993.
- [Curien, 1992] P.-L. Curien. Observable algorithms on concrete data structures. In *Proc. 7th IEEE Symp. Logic in Computer Science*, pages 432–443. IEEE Computer Society Press, 1992.
- [Davey and Priestley, 1990] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge Mathematical Textbooks. Cambridge University Press, 1990.
- [Ehrhard, 1994a] T. Ehrhard. Hypercoherences: a strongly stable model of linear logic, 1994. Preprint.
- [Ehrhard, 1994b] T. Ehrhard. Projecting sequential algorithms on strongly stable functions. *Journal of Pure and Applied Logic*, To appear, 1994.
- [Felleisen and Friedman, 1986] M. Felleisen and D. P. Friedman. Control operators, the SECD-machine, and the  $\lambda$ -calculus. In M. Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217. Elsevier Science Publishers, Amsterdam, 1986.
- [Gandy, 1993] R. O. Gandy. Dialogues, Blass games, sequentiality for objects of finite type. Unpublished manuscript, 1993.
- [Girard, 1972] J.-Y. Girard. *Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur*. PhD thesis, Université de Paris, 1972.



- [Girard, 1981] J.-Y. Girard.  $\pi_1^2$  Logic: Part I, Dilators. *Annals of Mathematical Logic*, pages 75–219, 1981.
- [Girard, 1986] J.-Y. Girard. The system  $F$  of variable types, fifteen years later. *Theoretical Computer Science*, 45:159–192, 1986.
- [Girard, 1987] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.
- [Girard et al., 1989] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science 7. Cambridge University Press, 1989.
- [Gödel, 1958] K. Gödel. Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes. *Dialectica*, pages 280–287, 1958.
- [Gödel, 1990] K. Gödel. *Kurt Gödel collected works, volumes I and II*. Oxford University Press, Oxford, 1990.
- [Gunter and Scott, 1990] C. A. Gunter and D. S. Scott. Semantic domains. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 635–674. Elsevier, 1990.
- [Gunter, 1992] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. MIT Press, 1992.
- [Jim and Meyer, 1991] T. Jim and A. R. Meyer. Full abstraction and the context lemma. In Ito and Meyer, editors, *Proc. Int. Conf. Theoretical Aspects of Computer Software*, pages 131–151. LNCS Vol. 526, Springer-Verlag, 1991.
- [Jung and Stoughton, 1993] A. Jung and A. Stoughton. Studying the fully abstract model of PCF within its continuous function model. In *Proc. Int. Conf. Typed Lambda Calculi and Applications, Utrecht, March, 1993*, pages 230–245. LNCS Vol. 664, Springer-Verlag, 1993.
- [Kahn and Plotkin, 1978] G. Kahn and G. D. Plotkin. Structures de données concrètes. Technical Report 338, INRIA-LABORIA, 1978.
- [Kahn, 1988] G. Kahn. Natural semantics. In K. Fuchi and M. Nivat, editors, *Programming of Future Generation Computers*, pages 237–259. Elsevier Science Publisher, BV, (North-Holland), 1988.
- [Kahn and Plotkin, 1993] G. Kahn and G. D. Plotkin. Concrete domains. *Theoretical Computer Science*, (Böhm Festschrift Special Issue), 1993.
- [Kelly, 1982] G. M. Kelly. *Basic Concepts of Enriched Category Theory*. L. M. S. Lecture Notes Series 64. Cambridge University Press, 1982.
- [Kleene, 1959] S. C. Kleene. Recursive functionals and quantifiers of finite types I. *Transactions of the American Mathematical Society*, 91:1–52, 1959.
- [Kleene, 1963] S. C. Kleene. Recursive functionals and quantifiers of finite types II. *Transactions of the American Mathematical Society*, 108:106–142, 1963.

- [Kleene, 1978] S. C. Kleene. Recursive functionals and quantifiers of finite types revisited I. In J. E. Fenstad, R. O. Gandy, and G. E. Sacks, editors, *General Recursion Theory II, Proceedings of the 1977 Oslo Symposium*, pages 185–222. North-Holland, 1978.
- [Kleene, 1980] S. C. Kleene. Recursive functionals and quantifiers of finite types revisited II. In J. Barwise, H. J. Keisler, and K. Kunen, editors, *The Kleene Symposium*. North-Holland, 1980.
- [Kleene, 1982] S. C. Kleene. Recursive functionals and quantifiers of finite types revisited III. In G. Metakides, editor, *Patras Logic Symposium*. North-Holland, 1982.
- [Kleene, 1985] S. C. Kleene. Recursive functionals and quantifiers of finite types revisited IV. In *Proc. Symposia in Pure Mathematics, Vol. 42*. American Mathematical Society, 1985.
- [Lamarche, 1993] F. Lamarche. Stable domains are generalized topological spaces. *Theoretical Computer Science*, 111:103–123, 1993.
- [Lambek and Scott, 1986] J. Lambek and P. J. Scott. *Introduction to Higher Order Categorical Logic*. Cambridge Studies in Advanced Mathematics No. 7. Cambridge University Press, 1986.
- [MacLane, 1971] S. MacLane. *Categories for the Working Mathematician*. Graduate Text in Mathematics 5. Springer-Verlag, 1971.
- [Martin-Löf, 1979] P. Martin-Löf. Constructive mathematics and computer programming. In *International Congress for Logic, Methodology and Philosophy of Science*, pages 538–571. North-Holland, 1979.
- [Meyer and Cosmadakis, 1988] A. R. Meyer and S. C. Cosmadakis. Semantical paradigms: notes for an invited lecture. In *Proc. 3rd Annual IEEE Symp. Logic in Computer Science*. Computer Society Press, 1988.
- [Milner, 1975] R. Milner. Processes, a mathematical model of computing agents. In *Logic Colloquium, Bristol 1973*, pages 157–174. North-Holland, Amsterdam, 1975.
- [Milner, 1977] R. Milner. Fully abstract models of typed lambda-calculus. *Theoretical Computer Science*, 4:1–22, 1977.
- [Milner, 1989] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
- [Milner et al., 1990] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, Cambridge, MA, 1990.
- [Morris, 1968] J.-H. Morris. *Lambda Calculus Models of Programming Languages*. PhD thesis, MIT, 1968.
- [Mosses, 1990] P. Mosses. Denotational semantics. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 577–632. Elsevier, 1990.
- [Mulmuley, 1986] K. Mulmuley. Fully abstract submodels of typed lambda calculus. *Journal of Computer and System Sciences*, 33:2–46, 1986.

- [Mulmuley, 1987] K. Mulmuley. *Full Abstraction and Semantic Equivalence*. MIT Press, 1987.
- [Ong, 1988] C.-H. L. Ong. *The Lazy Lambda Calculus: An Investigation into the Foundations of Functional Programming*. PhD thesis, University of London, 1988. Also available as Cambridge University Computer Laboratory Technical Report No 259, June 1992.
- [Park, 1980] D. M. Park. Concurrency on automata and infinite sequences. In P. Deussen, editor, *Conference on Theoretical Computer Science*. LNCS Vol. 104, Springer-Verlag, 1980.
- [Pitts, 1994] A. M. Pitts. Computational adequacy via “mixed” inductive definitions. In *Proc. 18th Int. Symp. Mathematical Foundations of Computer Science, IX, New Orleans, 1993*. LNCS Vol. 802, Springer-Verlag, 1994.
- [Platek, 1966] R. A. Platek. *Foundations of Recursion Theory*. PhD thesis, Stanford University, 1966.
- [Plotkin, 1972] G. D. Plotkin. A set-theoretical definition of application. Technical Report MIP-R-95, School of A.I., University of Edinburgh, 1972.
- [Plotkin, 1976] G. D. Plotkin. A powerdomain construction. *SIAM Journal of Computing*, 5:452–487, 1976.
- [Plotkin, 1977] G. D. Plotkin. LCF as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.
- [Plotkin, 1981a] G. D. Plotkin. Cpo’s: Tools for making meanings, 1981. Post-Graduate Lecture Notes in Advanced Domain Theory, Dept. of Computer Science, University of Edinburgh.
- [Plotkin, 1981b] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Department of Computer Science, University of Aarhus, Denmark, 1981.
- [Plotkin, 1985] G. D. Plotkin. Types and partial functions, 1985. Post-Graduate Lecture Notes, Dept. of Computer Science, University of Edinburgh.
- [Sazonov, 1975] Y. Sazonov. Sequentiality and parallelly computable functionals. In *Proc. Symp. Lambda Calculus and Computer Science Theory*. LNCS Vol. 37, Springer-Verlag, 1975.
- [Sazonov, 1976] Y. Sazonov. Expressibility of functions in Scott’s LCF language (translated from Russian). *Algebra i Logika*, 15:308–330, 1976.
- [Scott, 1969] D. S. Scott. A type theoretical alternative to CUCH, ISWIM and OWHY. Unpublished handwritten manuscript, 1969.
- [Scott, 1993] D. S. Scott. A type-theoretical alternative to CUCH, ISWIM and OWHY. *Theoretical Computer Science* (Böhm Festschrift, Special Issue), 121:411–440, 1993.

- [Sieber, 1992] K. Sieber. Reasoning about sequential functions via logical relations. In M. P. Fourman *et al.*, editors, *Applications of Categories in Computer Science*, pages 66–94. Cambridge University Press, 1992.
- [Spector, 1962] C. Spector. Provably recursive functionals of analysis: a consistency proof of analysis by an extension of principles formulated in current intuitionistic mathematics. In *Recursive Function Theory, Proc. Symposia in Pure Mathematics V*, pages 1–27. American Mathematical Society, Providence, RI, 1962.
- [Statman, 1985] R. Statman. Logical relations and the typed  $\lambda$ -calculus. *Information and Control*, 65:85–97, 1985.
- [Stoughton, 1988] A. Stoughton. *Fully Abstract Models of Programming Languages*. Research Notes in Theoretical Computer Science. Pitman, 1988.
- [Stoughton, 1991a] A. Stoughton. Equationally fully abstract models of PCF. In *Proc. 5th Int. Conf. Mathematical Foundations of Programming Semantics*, pages 271–283. LNCS Vol. 442, Springer-Verlag, 1991.
- [Stoughton, 1991b] A. Stoughton. Interdefinability of parallel operations in PCF. *Theoretical Computer Science*, 79:357–358, 1991.
- [Stoughton, 1991c] A. Stoughton. Parallel PCF has a unique extensional model. In *Proc. 6th IEEE Annual Symp. Logic in Computer Science*, pages 146–151. IEEE Computer Society Press, 1991.
- [Tait, 1967] W. W. Tait. Intensional interpretation of functionals of finite type i. *Journal of Symbolic Logic*, 32:98–212, 1967.
- [Tarski, 1955] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.
- [Taylor, 1990] P. Taylor. An algebraic approach to stable domains. *Journal of Pure and Applied Algebra*, 64:171–203, 1990.
- [Vuillemin, 1974] J. Vuillemin. *Syntaxe, Sémantique et Axiomatique d'un Langage de Programmation Simple*. PhD thesis, Université Paris VII, 1974.
- [Wadsworth, 1976] C. P. Wadsworth. The relation between computational and denotational properties for Scott's  $D_\infty$ -models of the lambda calculus. *SIAM Journal of Computing*, 5:488–521, 1976.
- [Wadsworth, 1978] C. P. Wadsworth. Approximate reduction and lambda calculus models. *SIAM Journal of Computing*, 7:337–356, 1978.
- [Winskel, 1980] G. Winskel. *Events in Computation*. PhD thesis, University of Edinburgh, 1980.
- [Winskel, 1987] G. Winskel. Event structures. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Application and Relationships to Other Models of Concurrency*, pages 325–392. LNCS Vol. 255, Springer-Verlag, 1987.

[Winskel, 1993] G. Winskel. *The Formal Semantics of Programming Languages*. Foundations of Computing Series. MIT Press, 1993.



# Effective algebras

V. Stoltenberg-Hansen and J. V. Tucker

---

## Contents

1	Introduction . . . . .	358
	1.1 Computing in algebras . . . . .	359
	1.2 Examples of countable algebras . . . . .	361
	1.3 Examples of uncountable algebras . . . . .	363
	1.4 Definitions of computable algebras . . . . .	364
	1.5 General algebraic framework . . . . .	366
	1.6 Historical notes on computable algebra . . . . .	368
	1.7 Objectives and structure of the chapter . . . . .	371
	1.8 Prerequisites . . . . .	372
2	Computable algebras . . . . .	372
	2.1 Preliminaries on algebras . . . . .	373
	2.2 Computable, semicomputable, and cosemicomputable algebras . . . . .	377
	2.3 Invariance . . . . .	387
	2.4 A few computable constructions . . . . .	397
	2.5 Concluding remarks . . . . .	402
3	Algebraic characterisations of computable algebras . . . . .	402
	3.1 Computable data types and their specification . . . . .	403
	3.2 Equational specifications . . . . .	411
	3.3 Adequacy theorem . . . . .	421
	3.4 Equational specifications and term rewriting . . . . .	427
	3.5 Proof of First Completeness Theorem . . . . .	433
	3.6 Concluding remarks . . . . .	444
4	Domains and approximations for topological algebras . . . . .	445
	4.1 Approximation structures and topologies . . . . .	446
	4.2 Representing topological algebras by structured domains . . . . .	451
	4.3 Inverse limits and ultrametric algebras . . . . .	456
	4.4 Total elements in domains . . . . .	467
	4.5 Representability of locally compact Hausdorff algebras . . . . .	472
5	Effective domains . . . . .	479
	5.1 Basic theory . . . . .	480

5.2	Constructive subdomains . . . . .	485
5.3	Algebras effectively approximable by domains . . .	489
5.4	The Myhill–Shepherdson Theorem . . . . .	501
5.5	The Kreisel–Lacombe–Shoenfield Theorem . . . . .	505

## 1 Introduction

The theory of the computable functions establishes the scope and limits of computation by means of algorithms on the set  $\mathbb{N} = \{0, 1, 2, \dots\}$  of natural numbers. It is based on the assumption that this set of data, equipped with some very simple operations, forms an algebraic structure that is computable in some absolute sense: the data are concrete objects and the operations on the data are computable. An objective of the theory is to analyse and classify the functions and sets that can be defined, using a variety of models of computation and formal systems, over some fundamental algebras of natural numbers.

How can we explore the scope and limits of computation by means of algorithms on other sets of data? Some familiar examples of data are the integer, rational, real and complex numbers; matrices, polynomials and power series with these numbers as coefficients; points, lines, planes, and other geometric objects; finite and infinite strings of symbols; finite and infinite terms and trees; and the functions and sets associated with these data. If we organise data into algebraic structures the following questions can be addressed:

*Which algebras of data are computable? Which functions and sets can be computed on these algebras?*

*Which algebras of data can be approximated by computable algebras? Which functions and sets on these algebras have computable approximations?*

Effective algebra is a theory that provides answers for these questions and goes on to establish the scope and limits of computation by means of algorithms for *any* set of data. It achieves this by applying the theory of computation on  $\mathbb{N}$  to universal algebras. Thus, it classifies what data can be represented algorithmically, and what sets and functions can be defined by algorithms, in the same terms as those of the Church–Turing Thesis for algorithms on  $\mathbb{N}$ .

This chapter is an introduction to effective algebra. It describes some of the basic concepts and results about

*computable algebras;*

*computable approximations of topological algebras;*

and their applications to specific structures.

In the first part we introduce the study of computable algebras and examine the specification of computable algebras by equational specification

methods. Among the results we prove is that the computable algebras are precisely the algebras that are definable by finite sets of equations whose term rewriting systems are complete (i.e. Church–Rosser and strongly terminating).

In the second part we introduce the study of effective approximations of uncountable topological algebras. We present a new systematic method for representing large classes of topological algebras using domains. Then we apply the theory of effective domains to show how to approximate effectively ultrametric algebras and locally compact Hausdorff algebras.

Effective algebra addresses central concerns in mathematics and computer science, and has a long history and several subfields with deep results, such as the study of the word problem for groups.

The chapter is closely linked with the chapter on universal algebra (in Volume 1) and draws on other subjects that are covered in chapters of the Handbook including: computability (Volume 1), term rewriting (Volume 2), topology (Volume 1) and domain theory (Volume 3). Further information on prerequisites is given in Section 1.8.

## 1.1 Computing in algebras

In a simple form an algebra consists of a non-empty set

$$A,$$

called a *carrier*, together with a finite family

$$a_1, \dots, a_p$$

of elements of  $A$ , called *constants*, and a finite family

$$\sigma_1, \dots, \sigma_q$$

of total functions on  $A$  called *operations*; these functions are of the form

$$\sigma_i : A^{n(i)} \rightarrow A$$

where  $n(i)$  is the arity of the operation  $\sigma_i$  for  $1 \leq i \leq q$ . The structure is usually written

$$A = (A; a_1, \dots, a_p, \sigma_1, \dots, \sigma_q).$$

A set  $\Sigma$  of names for the data set, constants, and operations (and their arities) of the algebra  $A$  is called a *signature*.

More generally, an algebra can consist of a *family* of non-empty sets, constants from the different sets, and operations taking arguments from several sets; these are called *many sorted algebras*. Hence the simple type above is called a *single sorted algebra*. Further complications of the notion

arise if the number of sets, constants, or operations is not finite; if empty sets are allowed or the operations are partial; and if the arguments of the operations are not finite tuples but infinite tuples. Almost exclusively, we will use single sorted algebras with finite signatures in order to keep our discussion as simple as possible.

Algebras can be found throughout mathematics and computer science. We may caricature much of the history of algebra as the discovery and study of special kinds of data and operations, and especially the natural, integer, rational, real, and complex numbers. In the nineteenth century, algebras of functions, propositions, sets, and quaternions were investigated. Ideas and methods of abstraction were also developed that led to the algebraic theories of groups, rings, and fields, and Boolean algebras. The general notion and theory of single sorted algebras was first formulated and studied in Birkhoff [1933; 1935]. Today, algebras are central objects of study in the mathematical subjects of universal algebra (see Grätzer [1979] and Cohn [1981]) and, to a lesser extent, model theory (see Chang and Keisler [1990] and Hodges [1993]).

In computer science, these algebras have been used to provide a general theory of data. More specifically, they have been used to define and analyse many new forms of data types, classify data representations, and perform the modularisation of computing systems. There is a considerable literature available which may be accessed through survey works such as Meinke and Tucker [1992], Wechler [1991], and Wirsing [1990].

The first set of questions we will address in this chapter are:

*When is an algebra  $A$  computable? What functions on  $A$  are computable? What sets on  $A$  are decidable or, at least, semidecidable?*

The starting point for this investigation is the theory of the recursive functions on the set  $\mathbb{N} = \{0, 1, 2, \dots\}$  of natural numbers. According to the Church–Turing Thesis, the class of recursive functions of the form

$$f : \mathbb{N}^n \rightarrow \mathbb{N},$$

for  $n > 0$ , is precisely the class of functions definable by means of algorithms on the natural numbers. The theory of the recursive functions is a deep subject that provides a thorough account of many technical questions concerning computability and non-computability, and definability and provability in formal theories. It also provides simple answers to questions about computation with simple data representations and constructs. The theory is an abstraction of the digital view of computation, to be found in writings about general purpose computers, from Babbage's plans for the Analytical Engine (of 1835) to those of contemporary machines (see Babbage [1989]).

Assuming the Church–Turing Thesis, we may use the theory of the re-



cursive functions to delimit the functions implementable on digital computers. We will use the theory to give precise answers to the above questions about algebras, which may be translated into

*When is an algebra  $A$  implementable on a computer? What functions and sets on an algebra  $A$  can be implemented on a computer?*

Thus, we will give precise answers to the general question

*What sets of data and functions on that data can be implemented on a computer, in principle?*

These questions admit a number of answers. Our investigation will be in two parts: we will examine the ideas that

1.  $A$  is a computable algebra that is implementable on a computer; and
2.  $A$  is approximated by a computable structure that is implementable on a computer.

We will assume that for an algebra to be computable it must be countable, so that its elements may be enumerated. For an algebra to be approximable it must have a topological structure, so that its elements may be approximated, and it is likely to be uncountable. In this chapter we will devote Sections 2 and 3 to the case of countable algebras, and Sections 4 and 5 to the case of possibly uncountable topological algebras.

## 1.2 Examples of countable algebras

First, let us turn to the raw material of our subject, namely specific algebraic structures. We give a list of countable algebras and invite the reader to speculate on the algorithmic nature of some of the algebras in the list (it is not essential that the reader understand or recognise all the examples).

Let  $\mathbb{N}$ ,  $\mathbb{Z}$ ,  $\mathbb{Q}$ ,  $\mathbb{R}$ , and  $\mathbb{C}$  be the sets of natural, integer, rational, real, and complex numbers, respectively.

*Is this countable algebra computable, or effective, in some precise sense?*

1.  $(\mathbb{N}; 0, n+1)$ ;
2.  $(\mathbb{N}; 0, n+1, n+m, n \cdot m)$ ;
3.  $(\Omega; c_1, \dots, c_p, f_1, \dots, f_q)$  for  $\Omega \subseteq \mathbb{N}$  a recursive set,  $c_i \in \Omega$ , and  $f_j : \Omega^{n(j)} \rightarrow \Omega$  recursive functions;
4.  $(\mathbb{Z}; 0, 1, x+y, x-y, x \cdot y)$ ;
5.  $(\mathbb{Q}; 0, 1, x+y, x-y, x \cdot y)$ ;
6.  $(\mathbb{Q}; 0, 1, x+y, x-y, x \cdot y, x^{-1})$ ;
7. any finite semigroup of continuous functions of the form  $f : [0, 1] \rightarrow [0, 1]$  with composition as the operation;
8. any algebra  $(A; a_1, \dots, a_p, \sigma_1, \dots, \sigma_q)$  wherein  $A$  is a finite set;



9. the rings  $\mathbb{Z}[X_1, \dots, X_n]$  of all polynomials in  $n > 0$  indeterminates over the integers;
10. the fields  $\mathbb{Q}(X_1, \dots, X_n)$  of all rational functions in  $n > 0$  indeterminates over the rationals;
11. the ring  $\mathbb{Z}[X_1, X_2, \dots]$  of all polynomials in infinitely many indeterminates over the integers;
12. the field  $\mathbb{Q}(X_1, X_2, \dots)$  of all rational functions in infinitely many indeterminates over the rationals;
13. the subfield  $\mathbb{Q}(\sqrt{2})$  of the field of real numbers generated by  $\mathbb{Q}$  and  $\sqrt{2}$ ;
14. the subfield  $\mathbb{Q}(\pi)$  of the field of real numbers generated by  $\mathbb{Q}$  and  $\pi$ ;
15. the subfield  $\mathbb{Q}(r)$  of the field of real numbers generated by  $\mathbb{Q}$  and a non-computable real number  $r$ ;
16. the subfield  $\mathbb{A}$  of the field of complex numbers containing precisely the algebraic numbers;
17. the subfield  $\mathbb{A}_{\mathbb{R}}$  of the field of real numbers containing precisely the real algebraic numbers;
18. the subfield  $F$  of the field of real numbers generated by  $\mathbb{Q}$  and the set  $\{\sqrt{p} : p \text{ prime}\}$ ;
19. the set  $U(F)$  of all roots of unity in a computable subfield  $F$  of the complex numbers;
20. any countable field;
21. any finitely generated commutative ring;
22. the subalgebra  $A$  of  $(\mathbb{R}; 0, 1, x+y, -x, x \cdot y, x^{-1}, exp)$  generated by the set  $\mathbb{Q}$  of rationals;
23. the classical groups  $GL(n, \mathbb{Q})$ ,  $SL(n, \mathbb{Q})$ ,  $O(n, \mathbb{Q})$ , and  $L(n, \mathbb{Q})$  of matrices over the field  $\mathbb{Q}$ ;
24. the semigroup  $A^*$  of all finite strings over the finite or infinite alphabet  $A$ ;
25. the finitely presented semigroup  $\langle a, b, c, d, e; ac = ca, ad = da, bc = cb, bd = db, eca = ce, edb = de, cca = ccae \rangle$ ;
26. any finitely generated abelian group;
27. any solvable group;
28. any finitely presented group;
29. any subgroup of a finitely presented group;
30. any finitely generated group of matrices over any field;
31. any finitely generated subalgebra of an affine algebra, i.e. an algebra of the form  $\langle A; a_1, \dots, a_p, \sigma_1, \dots, \sigma_q \rangle$  where  $A$  is a subset of an affine space  $F^n$  and the operations are polynomial functions over the field  $F$ ;
32. the algebra  $T(\Sigma, X)$  of all terms over signature  $\Sigma$  in the set  $X$  of indeterminates;

33. the algebra  $CT_\infty(\Sigma, X)$  of all computable infinite terms over signature  $\Sigma$  in the set  $X$  of indeterminates;
34. the semigroup of primitive recursive functions of the form  $f : \mathbb{N} \rightarrow \mathbb{N}$ ;
35. the semigroup of partial recursive functions of the form  $f : \mathbb{N} \rightarrow \mathbb{N}$ ;
36. the semigroup of total recursive functions of the form  $f : \mathbb{N} \rightarrow \mathbb{N}$ ;
37. the field  $(\mathbb{R}_k; 0, 1, x + y, -x, x \cdot y, x^{-1})$  of recursive real numbers;
38. the algebra  $(\mathbb{R}_k; 0, 1, x + y, -x, x \cdot y, x^{-1}, \sqrt{x}, \sin, \cos, \exp)$  of recursive real numbers;
39. the classical groups  $GL(n, \mathbb{R}_k), SL(n, \mathbb{R}_k), O(n, \mathbb{R}_k)$ , and  $L(n, \mathbb{R}_k)$  of matrices over the recursive reals  $\mathbb{R}_k$ ;
40. the initial algebra of a set of equations or conditional equations defining a stack;
41. any initial algebra of a finite set of equations or conditional equations;
42. any initial algebra of an infinite set of equations or conditional equations;
43. the algebra  $A_\omega$  of finite processes satisfying the laws of *ACP* in Baeten and Weijland [1990];
44. the algebra of computable infinite processes in the projective limit model of *ACP*.

### 1.3 Examples of uncountable algebras

Each of the above algebras could be enumerated, at least. Now we consider the following list of uncountable algebras that cannot be enumerated, and we invite the reader to reflect on the following question. (Once again, it is not essential that the reader understand or recognise all the examples.)

*In what ways can each uncountable algebra be approximated in a computable or effective way?*

1.  $(\mathbb{R}; 0, 1, x + y, -x, x \cdot y, x^{-1})$ ;
2.  $(\mathbb{R}; 0, 1, \pi, x + y, -x, x \cdot y, x^{-1})$ ;
3.  $(\mathbb{R}; 0, 1, r, x + y, -x, x \cdot y, x^{-1})$  for any real number  $r$ ;
4.  $(\mathbb{R}; 0, 1, x + y, -x, x \cdot y, x^{-1}, \sqrt{x})$ ;
5.  $(\mathbb{R}; 0, 1, x + y, -x, x \cdot y, x^{-1}, \sin, \cos, \exp)$ ;
6.  $(\mathbb{R}; 0, 1, x + y, -x, x \cdot y, x^{-1}, f)$  where  $f : \mathbb{R} \rightarrow \mathbb{R}$  is continuous;
7.  $(\mathbb{R}; 0, 1, x + y, -x, x \cdot y, x^{-1}, f)$  where  $f : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$  is the step function  $f(x, r) = 0$  if  $x < r$  and  $f(x, r) = 1$  if  $x \geq r$  for any  $x, r \in \mathbb{R}$ ;
8. the many sorted algebra  $(\mathbb{R}, \mathbb{N}, [\mathbb{N} \rightarrow \mathbb{R}]; 0, 1, x + y, -x, 0, n + 1, eval)$  that adds infinite sequences or streams of real numbers to the additive abelian group of reals, the naturals with zero and successor, and the sequence evaluation map  $eval: [\mathbb{N} \rightarrow \mathbb{R}] \times \mathbb{N} \rightarrow \mathbb{R}$ ;
9.  $(\mathbb{C}; 0, 1, i, z + z', -z, z \cdot z', z^{-1})$ ;
10.  $(\mathbb{C}; 0, 1, i, z + z', -z, z \cdot z', z^{-1}, \bar{z}, |z|)$ ;

11. fractal subsets of  $\mathbb{C}$  such as the Mandelbrot and Julia sets;
12. the classical groups  $GL(n, \mathbb{R})$ ,  $SL(n, \mathbb{R})$ ,  $O(n, \mathbb{R})$ ,  $L(n, \mathbb{R})$  of matrices over  $\mathbb{R}$ ;
13. the circle group  $S^1$  and the torus group  $T^1$ ;
14. any compact Lie group;
15. the ring of formal power series  $\mathbb{Z}[[X]]$  over the integers  $\mathbb{Z}$  in indeterminate  $X$ ;
16. the ring of formal power series  $\mathbb{R}[[X]]$  over the reals  $\mathbb{R}$  in indeterminate  $X$ ;
17. the algebra  $T_\infty(\Sigma, X)$  of infinite terms over signature  $\Sigma$  and set  $X$  of indeterminates;
18. the infinite process algebra  $A^\infty$  model of  $ACP$ ;
19. the domain  $D_\infty$  that is a model of the  $\lambda$ -calculus;
20. the power domain  $P(\omega)$ ;
21. a universal domain  $U$ ;
22. the  $p$ -adic number field;
23. any complete local ring;
24. the semigroup of continuous functionals of the form  $F : [\mathbb{N} \rightarrow \mathbb{N}] \rightarrow [\mathbb{N} \rightarrow \mathbb{N}]$  with the operation of composition;
25. any algebra of complex numbers whose carrier and operations are first order definable over the field of complex numbers;
26. any algebra of real numbers whose carrier and operations are first order definable over the ordered field of real numbers.

## 1.4 Definitions of computable algebras

We will define and reflect on the notion of computable algebra that is at the heart of the theory of effective algebra. Computable algebras are countable; uncountable algebras will be approximated using computable algebras.

A computable algebra is an algebra that can be faithfully represented using the natural numbers in a recursive way.

**Definition 1.4.1 (First Definition).** An algebra

$$A = (A; a_1, \dots, a_p, \sigma_1, \dots, \sigma_q)$$

is *computable* if

1. the data of  $A$  can be *computably enumerated*: there exists a recursive subset  $\Omega \subseteq \mathbb{N}$  and a surjection

$$\alpha : \Omega \rightarrow A$$

that lists or enumerates, possibly with repetitions, all the elements of  $A$ ;

2. the operations of  $A$  are *computable in the enumeration*: for each operation  $\sigma_i : A^{n(i)} \rightarrow A$  of  $A$  there exists a recursive function

$$f_i : \Omega^{n(i)} \rightarrow \Omega$$

that *tracks* the  $\sigma_i$  in the set  $\Omega$  of numbers in the sense that for all  $x_1, \dots, x_{n(i)} \in \Omega$ ,

$$\sigma_i(\alpha(x_1), \dots, \alpha(x_{n(i)})) = \alpha(f_i(x_1, \dots, x_{n(i)}));$$

3. the equivalence of numerical representations of data in  $A$  is *decidable*: the equivalence relation  $\equiv_\alpha$  defined by

$$x_1 \equiv_\alpha x_2 \Leftrightarrow \alpha(x_1) = \alpha(x_2)$$

is recursive.

The computability of functions and sets over a computable algebra  $A$  may depend on the numbering  $\alpha$ ; thus, to be more precise, we will say that  $A$ , its functions and subsets etc. are  $\alpha$ -*computable*. Some fundamental theoretical questions are concerned with the stability or invariance of algorithmic properties of  $A$  across its various computable numberings.

Algebra provides a general theory that organises the design of data types and classifies the relationships between representations of data types. Thus, we can use algebra to isolate and refine some of the essential ideas in the First Definition, which involves two kinds of data: that from  $A$  and that from  $\mathbb{N}$ . Looking more closely at the definition that  $A$  is computable, we see, from the enumeration assumption (1) and tracking assumption (2), that we can construct an algebra of natural numbers of the same signature as  $A$ , namely

$$\Omega = (\Omega; c_1, \dots, c_p, f_1, \dots, f_q)$$

where the  $c_i \in \Omega$  and the  $f_j : \Omega^{n(j)} \rightarrow \Omega$  are total recursive functions. The tracking assumption (2) asserts that the enumeration function  $\alpha : \Omega \rightarrow A$  is a homomorphism from the number algebra  $\Omega$  to the data algebra  $A$ . Finally, for computability, we must assume that the homomorphism is decidable in the sense that its kernel  $\equiv_\alpha$  is a decidable set of pairs of numbers. Therefore, an equivalent form of the First Definition 1.4.1 is this:

**Definition 1.4.2 (Second Definition).** An algebra

$$A = (A; a_1, \dots, a_p, \sigma_1, \dots, \sigma_q)$$

is *computable* if there exists an algebra

$$\Omega = (\Omega; c_1, \dots, c_p, f_1, \dots, f_q)$$

of natural numbers with a recursive carrier set and recursive operations, and a surjective homomorphism

$$\alpha : \Omega \rightarrow A$$

with decidable kernel  $\equiv_\alpha$ .

By the Homomorphism Theorem for algebras, we know that

$$A \cong \Omega / \equiv_\alpha .$$

This definition of computability is a *finiteness condition* in algebra: (1) it is an isomorphism invariant, i.e. if  $A$  is computable and  $B$  is isomorphic to  $A$  then  $B$  is computable; and (2) it is a property that is possessed by all finite structures. Starting with these precise concepts we can define the intimately related notions of *semicomputable* and *cosemicomputable* algebras, the *computable functions* on and between algebras, and begin a comprehensive theory of the scope and limits of algorithms in algebra.

## 1.5 General algebraic framework for computations on algebras

As we have seen, an algebra is computable if it can be faithfully represented by recursive sets and functions of natural numbers; in particular, an algebra is implementable if it is implementable by the natural numbers.

It is possible to redesign the concept of computable algebra using other fundamental data sets. Recalling the analysis of computation in Turing [1936], we may choose to use strings over some alphabet of symbols. Against the background of Turing's detailed discussion of the limits of computation, we see that our definitions above leave open the precise representation of the natural numbers: we consider the natural numbers abstractly, independently of their decimal, binary, unary, or other notations. The assumption that computability can be adequately characterised by the natural numbers needs some discussion. In fact there are other candidates for fundamental data sets that are comparable with the natural numbers, for example the set of terms over a signature.

Thus, the strategy is to choose an algebra  $B$  of fundamental importance, say the natural numbers

$$(\mathbb{N}; 0, n + 1),$$

and choose a theory of computable sets and functions on it, say the recursion schemes of S. C. Kleene. Then we build algebras  $R$  that are computable with respect to  $B$ , and, finally, we throw the computability of  $R$  on to other algebras by means of numberings  $\alpha : R \rightarrow A$ .

Can we generalise our strategy to enlarge the study of the effectiveness of algebras in order to include the examples of uncountable algebras given in



Section 1.3? These algebras are not computable but have many algorithmic properties in need of precise analysis. For example, how do we study the effectiveness of the field

$$(\mathbb{R}; 0, 1, x + y, -x, x \cdot y, x^{-1})$$

of real numbers? Often, we can find several ways of studying a specific structure like that of the real numbers, but what we would like to have available are systematic methods, as general as those for computable algebras.

Let us consider more abstractly the structure of our notion of computable algebra as represented in the Second Definition 1.4.2.

**Definition 1.5.1.** Let  $R$  and  $A$  be algebras of signature  $\Sigma$ . We say that the algebra  $A$  is *represented* by the algebra  $R$  when there exists a surjective homomorphism  $\alpha : R \rightarrow A$ ; we call  $\alpha$  a *representation*. In these circumstances, we have

$$A \cong R / \equiv_{\alpha}$$

by the Homomorphism Theorem.

Thus, thinking in terms of the formulation corresponding with the First Definition 1.4.1, computation in  $A$  is represented by enumerating or encoding the data of  $A$  by  $R$  under the correspondence  $\alpha : R \rightarrow A$ , and computing in  $R$ .

To investigate algorithmic properties of more general classes of algebras we must consider either (i) more general models for computing exactly in an appropriate algebra  $R$  that represents the algebra  $A$  of interest, or (ii) general methods for computing approximately in such  $R$  with available models of computation.

### Exact computation

First, let us consider three ways in which we could compute *exactly* in, say, an uncountable representation algebra  $R$  by means of some generalised recursion theories (such as those described in Fenstad [1980]).

- (a) *Type 2 recursion theory.* Here we can replace the set  $\mathbb{N}$  and the total recursive functions on  $\mathbb{N}$  by the Baire space  $[\mathbb{N} \rightarrow \mathbb{N}]$  and the total recursive functionals on  $[\mathbb{N} \rightarrow \mathbb{N}]$ . This approach to effective algebra has been pursued in Weihrauch [1987]; see also Kreitz and Weihrauch [1984].
- (b) *Admissible recursion theory.* Here we can replace the set  $\mathbb{N}$  and the total recursive functions on  $\mathbb{N}$  by an admissible ordinal  $\alpha$  and the  $\alpha$ -recursive functions (see Sacks [1990]). Results known for computable rings and fields (in the sense above) can be lifted to this very general setting. For example, a result of Metakides and Nerode [1979] lifts to become the following: *Let  $F$  be a field with cardinality  $\kappa$  that is*

$\kappa$ -computable. Then the algebraic closure  $A$  of  $F$  is  $\kappa$ -computable and, furthermore,  $A$  is  $\kappa$ -computably unique if, and only if,  $F$  has a  $\kappa$ -computable splitting algorithm.

- (c) *Recursion theory over algebras.* Here we can replace the set  $\mathbb{N}$  and the total recursive functions on  $\mathbb{N}$  by any algebra  $B$  and the functions on  $B$  that are computable by means of finite deterministic models of computation, such as while-programs with arrays. This generalisation is closely connected with the classical theory of the recursive functions and has a large literature (see, for example, Shepherdson [1985], Moldestad *et al.* [1980a; 1980b], Tucker [1980b; 1991], Tucker and Zucker [1988; 1991; 1992; 1994]).

Although much is known about these three recursion theories, little is known of this exact approach to analysing computation in algebras.

### Approximate computation

Now we consider the idea that we do not generalise the recursion theory on  $\mathbb{N}$  to classify  $R$  but rather we investigate the computable approximation of  $R$ , and hence the approximation of  $A$  by computable structures. This approach is to be preferred because we desire an analysis of computation that is in the same terms as those of the Church–Turing Thesis.

We suppose that  $A$  is a topological algebra, i.e. an algebra whose carrier set is a topological space and whose operations are continuous. If  $R$  represents  $A$  then we will assume that  $R$  is a topological algebra and that the representation map  $\alpha : R \rightarrow A$  is continuous. How do we computably approximate  $R$ ?

We imagine building  $R$  from a set  $P$  of approximating data that is part of a computable structure. Each datum in  $R$  is approximated by some sequence  $(a_i)_{i \in I}$  of data from  $P$ . More specifically,  $R$  is a topological space obtained from  $P$  by some form of completion process. Another key feature of this approach is that some of the approximating sequences are computable, since  $P$  is computable. Let  $R_k$  be the set of elements in  $R$  that are computably approximable. This set is the basis of the computable approximation of  $R$  and hence of  $A$ .

We will use this general form of approximation of algebras  $R$ . We consider a special type of approximating structure  $P$  called a *conditional upper semilattice (cusi)* and a completion process called *ideal completion*. This process yields a *domain*. We will show that this particular method captures a large class of examples.

## 1.6 Historical notes on computable algebra

The systematic study of computable algebra can be said to originate from the work of A. Fröhlich and J. C. Shepherdson on effective operations in field theory. The paper by Fröhlich and Shepherdson [1956] is an important study of rings and fields that introduces many of the basic notions and

results on computable field extensions. The definition of an effective ring that they use rests on coding structures by symbols and computing with Turing machines.

The move to the enumeration of arbitrary sets using natural numbers was made in Rabin [1960] and Mal'cev [1961]. The idea is an obvious generalisation of the Gödel numbering of logical syntax. M. O. Rabin proved several results about computable groups, rings, and fields. For example, Fröhlich and Shepherdson established the computability of algebraic closures using the Steinitz construction (Steinitz [1910]) and the fundamental notion of *canonical (computable) field extension*, which allowed them to prove that *a computable field with a splitting algorithm has a computable algebraic closure which is computably unique*. A splitting algorithm for a field  $F$  is an algorithm that decides whether or not a polynomial over  $F$  is irreducible. Rabin established that the algebraic closure of any computable field is computable, using Artin's construction of the algebraic closure. A thorough mathematical account of computable rings and fields and their numberings is planned for Stoltenberg-Hansen and Tucker [1995].

A. I. Mal'cev studied computable universal algebras of the form given above (First Definition 1.4.1). Most of the notions we use are adaptations of those of Mal'cev who developed the theory of numberings, numbered sets, and numbered structures in a series of papers (see his selected works Mal'cev [1971]). Thorough mathematical accounts of computable sets, algebras, and their numberings are to be found in Ershov [1973; 1975; 1977b; 1979].

The theory of computable many sorted universal algebras begins in papers by J. A. Bergstra and J. V. Tucker starting in 1979: see their references and also Meseguer and Goguen [1985], Meseguer, Moss and Goguen [1992] and Wirsing [1990]. The motivation for this theory is to establish the scope and limits of algebraic specification methods for abstract data types. Some of this material is described in Section 3.

The origins of computable algebra cannot be separated from the origins of abstract algebra. The origins of abstract algebra are intertwined with the origins of programming in the scientific work of Charles Babbage. The history is in need of a great deal of difficult research and is far too complicated to attempt to describe here. However, we will mention some of the topics that must be examined in writing a history of our subject and of which the reader should be aware.

The concern for explicit constructions in algebra is long-standing. Problems on the nature of algebra arose in connection with the manipulation of data such as complex numbers. Discussions of the fundamental role of symbols and rules were advanced in works such as George Peacock's *Treatise on Algebra* of 1830 (Peacock [1830]). The algorithmic nature of algebra is important in the work of Charles-François Sturm on the number of roots of equations of 1835 (Sturm [1835]) and, of course, in Ada Lovelace's



notes on programming of 1845 (see Babbage [1989]). L. Kronecker was engaged in the development of algebraic systems in the 1850's and was particularly interested in explicit constructions (see Kronecker [1882] and Novy [1973, pp.131–150]). Important changes occurred as infinitistic and non-explicit methods were used in algebra through the contributions of Richard Dedekind and others. There is also Hilbert's celebrated solution to Gordon's Problem in 1880 (which developed into the Basis Theorem) and which is said to have established that algebra need not be limited to explicit constructions (see Reid [1970]).

In the twentieth century, Hilbert's address in 1900 contained algorithmic problems of note (e.g. the Tenth Problem). Arising from problems in geometry and the emerging subject of algebraic topology, the discovery of group presentations, by Walther von Dyck in 1882, and of the word, conjugacy, and related problems for group presentations, by Max Dehn in 1910–1911, are landmarks in the history of combinatorial group theory: see Dehn [1910; 1911], his selected works, [Dehn, 1987], and the historical studies of Chandler and Magnus [1982] and Wüßing [1984]. There is also the independent work of Axel Thue on the word problem for combinatorial systems akin to semigroup presentations (see Thue [1914] and his selected works Thue [1977]).

The conception and development of intuitionistic thinking by L. E. J. Brouwer is a significant historical landmark because of its influence on research on foundations (see the collected works of Brouwer [1975]).

Some early papers on constructive aspects of fields are van der Waerden [1930a] and Vandiver [1936]. In 1930 B. L. van der Waerden published his textbook *Moderne Algebra* which is a convenient landmark for the shape of contemporary algebra. In the early editions, there is a section on effective processes in field theory. Work on polynomials is later taken up in Krull [1953a; 1953b; 1954] and especially Fröhlich and Shepherdson [1956].

Constructive analyses of the Hilbert Basis Theorem and its generalisations are somewhat tricky and messy: see, for example, Seidenberg [1971; 1974a; 1974b] and, for a handy summary of subsequent results, Bridges and Richman [1987]. A detailed account of the computability of polynomial rings, rings with chain conditions, coherence etc. will be given in Stoltenberg-Hansen and Tucker [1995].

A historical account of the subject must also include the subsequent development of the theory of the recursive functions from 1936, motivated by research in logic and the foundations of mathematics and the emergence of undecidable problems (see Kleene [1981] and Gandy [1988]). The application of recursion theory has resulted in a vast literature containing results on decision problems in mathematics, logic, and computer science, and a tiny literature on decision problems in physics and biology (see the bibliography Hinman [1987]). The study of recursive analysis is particularly important to the development of effective algebra (see Goodstein [1961]

and Pour-El and Richards [1989]).

The generalisation of recursion theory from the set  $\mathbb{N}$  of natural numbers to functions and other objects of higher type over  $\mathbb{N}$  is relevant to effective algebra. This work of S. C. Kleene and others is the basis for the abstract development of the theory of domains by D. S. Scott and Y. Ershov.

The development of logical theories of constructive existence and of intuitionistic algebra is also necessary to examine to complete the picture: see Troelstra [1988; 1991] and Beeson [1985]).

## 1.7 Objectives and structure of the chapter

Effective algebra encompasses a wide range of subjects, some with highly developed technical theories of their own, such as those of group-theoretic decision problems, computable ring and field theory, and recursive analysis. In this short introduction to effective algebra we have chosen to emphasise (i) general algebras, rather than specific types of algebras, and (ii) topics that are of use to computer science.

We see algebra as providing a general theory of data that is theoretically satisfying and practically useful. Clearly, the theories of computable and computably approximable algebras are important for a general theory of data.

In meeting the aim of (ii) we have chosen to discuss the theory of equational specification of data types and the theory of effective domains. The study of computably approximating algebras is underdeveloped in computer science, yet it is a very important subject. The material on the effective representation of topological algebra by domains is new.

The chapter consists of two parts. The first part is devoted to the computability of countable algebras and the second to computably approximating, possibly uncountable topological algebras.

In Section 2 we define computable, semicomputable, and cosemicomputable algebras and the notions of computable subalgebra, congruence, factor algebra, homomorphism etc. We study numberings and their equivalence, and the extent to which computability is tied to numberings.

In Section 3 we consider the equational specification of algebras and the idea that semicomputable and computable algebras are characterised by initial algebra semantics.

In Section 4 we look at the foundations of an approximation theory for topological algebras. We explain new techniques based on the use of domains.

In Section 5 we develop the theory of effective domains and apply this to effectively approximate topological algebras.

There are many subjects that we have omitted from the discussion, and there is much good work by many authors that we have neglected to mention, often from ignorance or forgetfulness. We will be pleased to receive information and suggestions. Effective algebra is a subject that



offers its students considerable theoretical depth and scope, many areas of application, and scientific longevity. We hope this chapter provides an introduction that is satisfying, stimulating, and pleasurable.

We thank U. Berger (LMU, Munich), J. A. Bergstra (Amsterdam), Y. Ershov (Novosibirsk), E. Palmgren and J. Blanck (Uppsala), K. Stephenson (Swansea), and R. Thomas (Leicester) for useful information, comments or discussions on aspects of this work. We also thank our students at Uppsala and Swansea for their helpful responses to the material. We have benefited greatly from the award of an SERC Visiting Fellowship for V. Stoltenberg-Hansen to work at Swansea during the academic year 1993–1994. We are deeply grateful to Jane Spurr for her excellent and essential work in producing the final version of the chapter.

## 1.8 Prerequisites

First, we assume the reader is familiar with the theory of the recursive functions on the natural numbers. An introduction to the subject is contained in this Handbook (Volume 1, Chapter 2) and is treated in many books such as the textbooks Cutland [1980], Mal'cev [1970] and Rogers [1967].

Secondly, we assume the reader is familiar with the basics of universal algebra. Some mathematical textbooks are: Burris and Sankappanavar [1981] and McKenzie *et al.* [1987]. An introduction to the subject with the needs of computer science in mind is contained in this Handbook (Volume 1, Chapter 3). Some of its subject matter is also covered in Wechler [1991].

The application of universal algebra to the specification of data types is treated in Goguen, Thatcher and Wagner [1978], Ehrig and Mahr [1985], Meseguer and Goguen [1985] and Wirsing [1990].

Some familiarity with term rewriting will be useful and introductions are contained in this Handbook (Volume 2, Chapter 1) and in Dershowitz and Jouannaud [1990].

Fifthly, we will need some topology. This is covered in many books, such as Dugundji [1966] and Kelley [1955], and in a chapter in this Handbook (Volume 1, Chapter 5).

Finally, we assume the reader is familiar with a little domain theory. An account of this is given in Stoltenberg-Hansen *et al.* [1994]; and also in the chapter in this Handbook (Volume 3).

## 2 Computable algebras

The idea of a computable algebra is one of the fundamental theoretical concepts of computer science and mathematics. Through the applications of computer science and mathematics, the concept is destined to become a theoretical idea in other fields of science. The primary technical definitions of computable, semicomputable, and cosemicomputable algebras are derived from M. O. Rabin [1960] and, in particular, A. I. Mal'cev [1961], in-

dependent papers devoted to founding a general theory about computable algebraic systems and their computable morphisms. These papers are inspired by important work on computable fields in A. Fröhlich and J. C. Shepherdson [1956], and by an established theory of decidable and undecidable word problems for groups and semigroups: recall our historical notes in Section 1.6.

In this section we will mention only those ideas and facts which contribute to an understanding of the notion of computable algebra, or to proofs of theorems later on. We will explain basic ideas about *computable algebras*, *computable homomorphisms*, *computable subalgebras*, *computable congruences*, *computable factor algebras*, and so on. An important topic is the invariance of computable sets and functions under transformations of numberings and algebras; this is the subject of Section 2.3. We will not exhibit the theory of computable algebras, nor give illustrations of its depth and power through applications to specific topics, such as semigroups and groups, fields, Boolean algebras, or process algebras. Our modest aim for this section is to give a revised and updated treatment of some of the material in the early papers of A. I. Mal'cev. This will provide the reader with a good grasp of the fundamental idea of a computable algebra.

First, we establish some basic algebraic notations.

## 2.1 Preliminaries on algebras

The concepts, notations, and results for universal algebra that we need are widely used and are in Meinke and Tucker [1992] and Wechler [1991], for instance.

**Definition 2.1.1 (Signatures).** A *single sorted signature*  $\Sigma$  consists of a sort name  $s$ , and a family

$$\langle \Sigma_k : k \in \mathbb{N} \rangle$$

of sets, where each element  $c$  of  $\Sigma_0$  is called a *constant symbol* (of sort  $s$ ); and each element  $\sigma$  of  $\Sigma_k$  is called a *k-ary function symbol* (of type  $s^k \rightarrow s$ ). We will usually drop the qualification 'single sorted' and use the term *signature*.

A signature  $\Sigma$  is *finite* if each  $\Sigma_k$  is a finite set, and all but finitely many  $\Sigma_k = \emptyset$ , for  $k = 0, 1, 2, \dots$

A signature  $\Sigma$  is said to be *non-void*, *instantiated*, or *sensible* if  $\Sigma_0 \neq \emptyset$ .

Typically, a finite signature is written as a list of symbols

$$\langle s; c_1, \dots, c_p, \sigma_1, \dots, \sigma_q \rangle$$

where  $c_i \in \Sigma_0$  and  $\sigma_j \in \Sigma_{k(j)}$  for  $1 \leq i \leq p$  and  $1 \leq j \leq q$ ; it is often displayed as follows:

<b>signature</b>	$\langle name \rangle$
<b>sort</b>	$s$
<b>constants</b>	$c_1 : \rightarrow s$
	$\dots$
	$c_p : \rightarrow s$
<b>operations</b>	$\sigma_1 : s^{k(1)} \rightarrow s$
	$\dots$
	$\sigma_q : s^{k(q)} \rightarrow s$
<b>end</b>	

Unless stated otherwise, we assume throughout the chapter that all signatures are finite.

Let  $\Sigma^1$  and  $\Sigma^2$  be signatures. Then  $\Sigma^1$  is a *subsignature* of  $\Sigma^2$  if their sort names  $s_1$  and  $s_2$  are the same, and the constant and operation symbols of  $\Sigma^1$  are contained in  $\Sigma^2$ , i.e. for each  $k \in \mathbb{N}$ ,

$$\Sigma_k^1 \subseteq \Sigma_k^2.$$

We write  $\Sigma^1 \subseteq \Sigma^2$ .

**Definition 2.1.2 ( $\Sigma$ -algebras).** Let  $\Sigma$  be a single sorted signature. A  $\Sigma$ -algebra consists of a non-empty set

$$A$$

called the *carrier*, together with a family

$$\Sigma_0^A = \langle c_A \in A \mid c \in \Sigma_0 \rangle$$

of elements of  $A$  corresponding to the constant symbols, and a family

$$\Sigma_k^A = \langle \sigma_A \mid \text{for } \sigma \in \Sigma_k, \sigma_A : A^k \rightarrow A \rangle$$

of functions corresponding to the  $k$ -ary operation symbols, for  $k = 1, 2, \dots$

For a finite signature  $\Sigma = \langle s; c_1, \dots, c_p, \sigma_1, \dots, \sigma_q \rangle$  we often list the elements and functions that interpret its symbols as constants and operations of the algebra; for example, suppose that  $c_i$  is interpreted by  $a_i$  and  $\sigma_j$  is interpreted by  $f_j$ ; then  $\Sigma$  is interpreted by

$$A = (A; a_1, \dots, a_p, f_1, \dots, f_q).$$

The class of all  $\Sigma$ -algebras is denoted  $Alg(\Sigma)$ .

Definitions of  $\Sigma$ -subalgebra,  $\Sigma$ -congruence,  $\Sigma$ -factor algebra,  $\Sigma$ -homomorphism,  $\Sigma$ -isomorphism, and standard results about them can be found in the references.

**Definition 2.1.3 (Reducts and expansions).** Let  $A$  be a  $\Sigma$ -algebra and  $\Sigma_0 \subseteq \Sigma$ . Then the *reduct*  $A|_{\Sigma_0}$  of  $A$  by  $\Sigma_0$  is the  $\Sigma_0$ -algebra obtained from  $A$  by allowing as constants and operations only those constants and operations of  $A$  that are named in  $\Sigma_0$ .

The  $\Sigma$ -algebra  $A$  is an *expansion* of the  $\Sigma_0$ -algebra  $B$  if  $B = A|_{\Sigma_0}$ .

Let  $B$  be a subset of the  $\Sigma$ -algebra  $A$ . Then we define  $\langle B \rangle_{\Sigma_0}$  to be the smallest  $\Sigma_0$ -subalgebra of  $A|_{\Sigma_0}$  that contains  $B$ . We write  $\langle B \rangle$  for  $\langle B \rangle_{\Sigma}$ .

**Definition 2.1.4 (Minimal algebras).** A  $\Sigma$ -algebra is *minimal* if it contains no proper subalgebras. Equivalently, if  $\Sigma$  is non-void then  $A$  is minimal if the subalgebra  $\langle \emptyset \rangle$  generated by the empty set, or by the constants named in the signature, is  $A$ .

If  $K$  is a class of  $\Sigma$ -algebras then  $\text{Min}(K)$  is the subclass of all minimal  $\Sigma$  algebras in  $K$ .

**Definition 2.1.5 (Terms and term algebras).** Let  $X$  be a set, the elements of which are called *variables* or *indeterminates*. The set  $T(\Sigma, X)$  of all  $\Sigma$ -terms in the variables of  $X$  is inductively defined by

1. for  $x \in X$ ,  $x \in T(\Sigma, X)$ ;
2. for  $c \in \Sigma_0$ ,  $c \in T(\Sigma, X)$ ; and
3. for  $\sigma \in \Sigma_k$  and  $t_i \in T(\Sigma, X)$ ,  $\sigma(t_1, \dots, t_k) \in T(\Sigma, X)$ .

We define the *term algebra*  $T(\Sigma, X)$  of all  $\Sigma$ -terms in the variables of  $X$  as follows. The carrier of the term algebra is  $T(\Sigma, X)$ . The constants of  $T(\Sigma, X)$  are defined by: for  $c \in \Sigma_0$ ,

$$c_{T(\Sigma, X)} = c.$$

The operations are defined by: for  $\sigma \in \Sigma_k$ ,

$$\sigma_{T(\Sigma, X)}(t_1, \dots, t_k) = \sigma(t_1, \dots, t_k)$$

for  $t_1, \dots, t_k \in T(\Sigma, X)$ .

If  $X \subseteq Y$  then  $T(\Sigma, X)$  is a  $\Sigma$ -subalgebra of  $T(\Sigma, Y)$ . We write  $T(\Sigma)$  when  $X = \emptyset$ ; the terms in  $T(\Sigma)$  are called *closed terms*.

**Definition 2.1.6 (Equations).** Let  $\Sigma$  be a signature,  $X$  a set of variables and  $x = (x_1, \dots, x_k)$  a list of variables from  $X$ . A  $\Sigma$ -equation is an expression  $e$  of the form

$$t(x) = t'(x)$$

where  $t(x), t'(x) \in T(\Sigma, X)$ . Let  $\text{Eqn}(\Sigma, X)$  be the set of all equations.

Let  $e, e' \in \text{Eqn}(\Sigma, X)$ . We say  $e$  and  $e'$  are *permutation equivalent* if there is a permutation of the set  $X$  of variables that transforms  $e$  to  $e'$ . If  $e$  and  $e'$  are permutation equivalent then we write  $e \equiv_{\text{perm}} e'$ . Let  $[e] = \{e' \in \text{Eqn}(\Sigma, X) \mid e' \equiv_{\text{perm}} e\}$ .

Let  $E$  be a set of equations. Define the *perm-closure*  $\acute{E}$  of  $E$  to be

$$\acute{E} = \cup_{e \in E} [e].$$

We say that  $E$  is *perm-closed* if  $E = \acute{E}$ .

**Definition 2.1.7 (Term evaluation).** An *assignment* of elements of an algebra  $A$  to the variables of  $X$  is map  $a : X \rightarrow A$ . Given any assignment  $a$  we define the *term evaluation map*

$$\nu_A(a) : T(\Sigma, X) \rightarrow A$$

by induction on terms:

$$\begin{aligned} \text{for } c \in \Sigma_0, \quad & \nu_A(a)(c) = c_A; \\ \text{for } x \in X, \quad & \nu_A(a)(x) = a(x); \\ \text{for } \sigma \in \Sigma_k, \quad & \nu_A(a)(\sigma(t_1, \dots, t_k)) = \sigma_A(\nu_A(a)(t_1), \dots, \nu_A(a)(t_k)). \end{aligned}$$

Clearly,  $\nu_A(a)$  is a homomorphism (by its definition on the constants and operations on  $T(\Sigma, X)$ ).

**Lemma 2.1.8.** *For any  $\Sigma$ -algebra  $A$  and any assignment map  $a$ , the term evaluation map  $\nu_A(a) : T(\Sigma, X) \rightarrow A$  is the only  $\Sigma$ -homomorphism from  $T(\Sigma, X)$  to  $A$  that extends the assignment map  $a$ .*

The lemma is the statement that  $T(\Sigma, X)$  is free on  $X$  for  $\text{Alg}(\Sigma)$ .

The image  $\text{im}(\nu_A(a))$  of  $\nu_A(a)$  is a  $\Sigma$ -subalgebra of  $A$ ; it is the  $\Sigma$ -subalgebra of  $A$  generated by the subset  $\{a(x) : x \in X\}$ .

Let  $\equiv_a$  be the kernel congruence of the homomorphism  $\nu_A(a)$  that is defined on  $A$  by

$$t \equiv_a t' \Leftrightarrow \nu_A(a)(t) = \nu_A(a)(t').$$

Thus, since  $\nu_A(a)$  is an epimorphism onto  $\text{im}(\nu_A(a))$ ,

$$T(\Sigma)/\equiv_a \cong \langle \{a(x) : x \in X\} \rangle.$$

Let the set of assignments be denoted  $A^X$ . Then we can define another form of term evaluation function

$$TE : T(\Sigma, X) \times A^X \rightarrow A$$

by  $TE(t, a) = \nu_A(a)(t)$ , for  $t \in T(\Sigma, X)$  and  $a \in A^X$ .

We use  $T(\Sigma, X)$  in the cases when  $X = \{x_1, x_2, \dots\}$ ,  $X = \{x_1, \dots, x_n\}$ , and also  $X = \emptyset$ .

Suppose the signature  $\Sigma$  is non-void. If  $X = \emptyset$  then  $T(\Sigma)$  is non-empty and there are no assignments so we write

$$\nu_A : T(\Sigma) \rightarrow A.$$



The above lemma is worth repeating for  $X = \emptyset$ ; it is the statement that  $T(\Sigma)$  is initial for  $\text{Alg}(\Sigma)$ :

**Lemma 2.1.9.** *Suppose the signature  $\Sigma$  is non-void. For any  $\Sigma$ -algebra  $A$  the map  $\nu_A : T(\Sigma) \rightarrow A$  is the only  $\Sigma$ -homomorphism from  $T(\Sigma)$  to  $A$ .*

Then the image of the map  $\nu_A$  is the subalgebra of  $A$  generated by the elements named as constants in the signature  $\Sigma$ ; hence  $A$  is  $\Sigma$ -minimal if, and only if,  $\nu_A$  is a surjection.

Let  $\equiv_A$  be the kernel congruence of homomorphism  $\nu_A$  that is defined on  $A$  by

$$t \equiv_A t' \Leftrightarrow \nu_A(t) = \nu_A(t').$$

Thus, if  $A$  is minimal,  $T(\Sigma)/\equiv_A \cong A$ .

Using  $\nu_A(a)$  we can define the idea of the validity  $A, a \models e$  of an equation  $e$  for an assignment  $a$  in an algebra  $A$ .

**Definition 2.1.10 (Equational theories).** Let  $E \subseteq \text{Eqn}(\Sigma, X)$ . The class of all algebras satisfying all the equations in  $E$  at all assignments we denote  $\text{Alg}(\Sigma, E)$ .

The subjects of equational logic, its soundness, and completeness are explained in the recommended reading.

## 2.2 Computable, semicomputable, and cosemicomputable algebras

First, we will define numberings, effective numberings, and computable, semicomputable, and cosemicomputable numberings.

**Definition 2.2.1 (Numberings).** Let  $A$  be an algebra of signature  $\Sigma$ . A *numbering* of  $A$  consists of a set  $\Omega_\alpha$  of natural numbers and a surjection  $\alpha : \Omega_\alpha \rightarrow A$  such that for each  $k$ -ary operation symbol  $\sigma \in \Sigma_k (k \geq 0)$  with corresponding  $k$ -ary operation  $\sigma_A$  of  $A$ , there exists a total *tracking function*  $f : \Omega_\alpha^k \rightarrow \Omega_\alpha$  such that for all  $x_1, \dots, x_k \in \Omega_\alpha$ ,

$$\sigma_A(\alpha(x_1), \dots, \alpha(x_k)) = \alpha(f(x_1), \dots, f(x_k));$$

or, equivalently,  $f$  commutes the following diagram,

$$\begin{array}{ccc} A^k & \xrightarrow{\sigma_A} & A \\ \alpha^k \uparrow & & \uparrow \alpha \\ \Omega_\alpha^k & \xrightarrow{f} & \Omega_\alpha \end{array}$$

wherein  $\alpha^k(x_1, \dots, x_k) = (\alpha(x_1), \dots, \alpha(x_k))$  for  $x_1, \dots, x_k \in \Omega_\alpha$ . In the case that  $k = 0$  the tracking operations are codes for the constants.

We combine the set  $\Omega_\alpha$ , the numbers from  $\Omega_\alpha$  that label the constants, and the tracking functions, to constitute a  $\Sigma$ -algebra  $R_\alpha$  of natural numbers. Then a numbering is a  $\Sigma$ -epimorphism  $\alpha : R_\alpha \rightarrow A$ .

Consider the kernel relation  $\equiv_\alpha$  on the number algebra  $R_\alpha$ , defined for  $x, y \in \Omega_\alpha$  by

$$x \equiv_\alpha y \text{ if, and only if, } \alpha(x) = \alpha(y) \text{ in } A.$$

The relation is a  $\Sigma$ -congruence on  $R$ . By the Homomorphism Theorem,

$$A \cong R / \equiv_\alpha.$$

Quite generally, a numbering of  $A$  is simply an epimorphism from any algebra of natural numbers to  $A$ ; and a numbered algebra is a homomorphic image of an algebra of natural numbers.

We often denote a numbering by  $\alpha : \Omega_\alpha \rightarrow A$  or simply  $\alpha$ . But if  $\Sigma$  is a signature with  $p$  constant symbols and  $q$  operation symbols then the numbering consists of the map  $\alpha$  together with

$$(\Omega_\alpha, \equiv_\alpha, c_1, \dots, c_p, f_1, \dots, f_q).$$

**Definition 2.2.2 (Effective numberings).** An *effective numbering*  $\alpha$  of  $A$  is a numbering that consists of a recursive set  $\Omega_\alpha$  of natural numbers and for each  $k$ -ary operation  $\sigma_A$  of  $A$  a recursive tracking function  $f : \Omega_\alpha^k \rightarrow \Omega_\alpha$ .

Thus, an effective numbering  $\alpha$  is simply a  $\Sigma$ -epimorphism from a recursive algebra of natural numbers; and an effectively numbered algebra is simply a homomorphic image of a recursive algebra of natural numbers.

By a recursive function  $f : \Omega_1 \rightarrow \Omega_2$  we mean a partial recursive function  $f : \mathbb{N} \rightarrow \mathbb{N}$  such that  $\Omega_1 \subseteq \text{dom}(f)$  and  $f(\Omega_1) \subseteq \Omega_2$ . If  $\Omega_1$  is recursive we can take  $f$  to be a total recursive function.

We now give the primary definitions of this chapter.

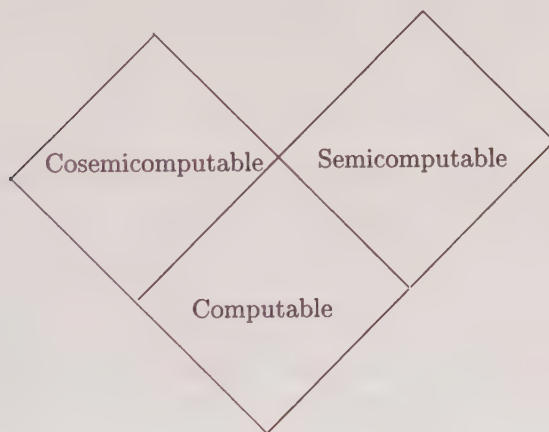
**Definition 2.2.3 (Computable numberings).** Let  $A$  be an algebra and  $\alpha : \Omega_\alpha \rightarrow A$  an effective numbering for  $A$ . Then:

1. The numbering  $\alpha$  is called a *computable numbering* if, and only if, the relation  $\equiv_\alpha$  is recursive on the recursive set  $\Omega_\alpha$ ; in this case the algebra  $A$  is said to be *computable under*  $\alpha$ .
2. The numbering  $\alpha$  is called a *semicomputable numbering* if, and only if, the relation  $\equiv_\alpha$  is recursively enumerable on the recursive set  $\Omega_\alpha$ ; in this case the algebra  $A$  is said to be *semicomputable under*  $\alpha$ .
3. The numbering  $\alpha$  is called a *cosemicomputable numbering* if, and only if, the relation  $\equiv_\alpha$  is co-recursively enumerable on the recursive set  $\Omega_\alpha$ ; in this case the algebra  $A$  is said to be *cosemicomputable under*  $\alpha$ .

From the computability of numberings we define the computability of algebras in an obvious way.

**Definition 2.2.4 (Computable, semicomputable and cosemicomputable algebras).** An algebra is *computable*, *semicomputable*, or *cosemicomputable* if there exists a computable, semicomputable, or cosemicomputable numbering for the algebra, respectively.

If  $A$  is computable under  $\alpha : R \rightarrow A$  then it is also semicomputable and cosemicomputable under that numbering. However, if  $A$  is semicomputable under  $\alpha_1 : R_1 \rightarrow A$  and cosemicomputable under  $\alpha_2 : R_2 \rightarrow A$  then we may not conclude that there is some numbering  $\alpha : R \rightarrow A$  under which  $A$  is computable. For the set of numberings of interest the situation may be visualised as in Fig. 1.

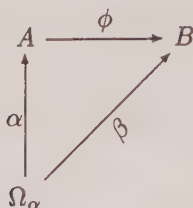


**Fig. 1.** The numberings

Let  $C(A)$ ,  $SC(A)$ , and  $CSC(A)$  be the set of all computable, semicomputable, and cosemicomputable numberings of  $A$ , respectively.

**Theorem 2.2.5.** Let  $A$  and  $B$  be algebras of signature  $\Sigma$ . Suppose that  $A$  is isomorphic with  $B$ . Then if  $A$  is computable, semicomputable, or cosemicomputable then  $B$  is computable, semicomputable, or cosemicomputable, respectively.

**Proof.** Let  $\alpha : \Omega_\alpha \rightarrow A$  be an effective numbering and let  $\phi : A \rightarrow B$  be an isomorphism. Define the map  $\beta = \phi \circ \alpha : \Omega_\alpha \rightarrow B$ , as shown in the diagram:



Since  $\alpha$  and  $\phi$  are epimorphisms their composition is an epimorphism and hence  $\beta$  is a numbering for  $B$ . Since the domain of  $\beta$  is the recursive algebra  $\Omega_\alpha$ , then  $\beta$  is an effective numbering. Consider the kernel of  $\beta$ : for  $x, y \in \Omega_\alpha$ ,

$$\begin{aligned}
 x \equiv_\beta y &\Leftrightarrow \beta(x) = \beta(y) \text{ in } B; \\
 &\Leftrightarrow \phi\alpha(x) = \phi\alpha(y) \text{ in } B; \\
 &\Leftrightarrow \alpha(x) = \alpha(y) \text{ in } A, \text{ since } \phi \text{ is an isomorphism;} \\
 &\Leftrightarrow x \equiv_\alpha y.
 \end{aligned}$$

Thus, the kernel of  $\beta$  coincides with that of  $\alpha$ . ■

The notion of numbering and effective numbering can be refined usefully by various logical and computational methods in order to analyse algebraic constructions that are not computable, or to measure more precisely the complexity of algebraic constructions that are computable. For example, the numerical sets, functions, and relations belonging to a numbering of an algebra can be classified using their definability in the arithmetical and other hierarchies, or even by their many-one degree or Turing degree of unsolvability. We have a use for numberings in which only the operations are recursive, and also for effective numberings in which the congruence is  $\Pi_2$  in the arithmetic hierarchy (see Section 5 on effective domains). Numberings classified by the hyperarithmetical and Grzegorczyk hierarchies have been used on groups in Feferman [1975] and Cannonito and Gatterdam [1973], respectively. In each case, as can be seen from the above proof, the desired new classification of numbered algebras and congruences is preserved by isomorphisms.

**Examples 2.2.6.** Let us consider some of the countable algebras in the list in 1.2. We will state which algebras are computable, but not necessarily in order; for reference we will give the number of the algebra in the list. It is not essential that the reader recognise all the examples or grasp all the comments. We will return to the examples at several places in the chapter.

1. The algebras of natural numbers (numbered 1–3 in the list in 1.2), integers (4), and rational numbers (5–6) are computable. The classical groups  $GL(n, \mathbb{Q})$ ,  $SL(n, \mathbb{Q})$ ,  $O(n, \mathbb{Q})$ , and  $L(n, \mathbb{Q})$  of matrices over the computable field  $\mathbb{Q}$  of rationals are computable (23).

2. Any finite algebra, no matter how complex its elements, is computable (7–8).
3. The polynomial rings and rational function fields with integer and rational number coefficients over the sets  $\{X_1, \dots, X_n\}$  and  $\{X_1, X_2, \dots\}$  of indeterminates are computable (9–12).
4. Any finite extension field  $\mathbb{Q}(a_1, \dots, a_n)$  of the field  $\mathbb{Q}$  of rationals is computable (13–15). More generally, any finite extension field of any computable field is computable. Any infinite transcendental extension field  $\mathbb{Q}(a_1, a_n, \dots)$  is computable. The infinite algebraic extension field  $\mathbb{Q}(\sqrt{p} : p \text{ prime})$  of  $\mathbb{Q}$  is computable (18).
5. The field  $\mathbb{A}$  of algebraic numbers is computable (16). More generally, the algebraic closure of any computable field is computable. The field  $\mathbb{A}_{\mathbb{R}}$  of real algebraic numbers is computable (17). More generally, the real closure of any computable ordered field is computable.
6. Let  $F$  be any field, not necessarily computable. Any finitely generated subalgebra of an affine algebra, i.e. an algebra of the form  $(A; a_1, \dots, a_p, \sigma_1, \dots, \sigma_q)$  where  $A$  is a subset of an affine space  $F^n$  and the operations are polynomial functions over the field  $F$ , is computable. For example, since the group  $GL(n, F)$  of non-singular matrices over  $F$  is an affine algebra, any finitely generated group of matrices over any field is computable (30–31).
7. Soon we will show that any countable field can be given an effective numbering but it need not possess a computable or even an arithmetical numbering (20). Any finitely generated commutative ring is computable (21); this follows from the Hilbert Basis Theorem and the decidability of the ideal membership problem in integer polynomial rings.
8. The semigroup  $A^*$  of all finite strings over the finite or countably infinite alphabet  $A$  is computable (24). But the finitely presented semigroup numbered (25) in the list is semicomputable but not computable.
9. Any finitely generated abelian group is computable; this is clear from the structure theorem for abelian groups (26). Generalisations of abelian groups, such as solvable groups, need not be computable (27).
10. Any finitely presented group is semicomputable and any finitely generated subgroup of a finitely presented group is semicomputable; neither type of group need be computable (28–29).
11. The semigroup of primitive recursive functions of the form  $f : \mathbb{N} \rightarrow \mathbb{N}$  is cosemicomputable but not computable (34). However, the semigroups of partial recursive functions and total recursive functions have effective numberings with arithmetically definable congruences (35–36).



12. Any initial algebra of a finite or recursively enumerable set of equations or conditional equations is semicomputable and it need not be computable. In fact the initial algebra of a finite set of equations or conditional equations defining a stack need not be computable (40–42). The algebra  $A_\omega$  of all finite processes satisfying the laws of *ACP* in Baeten and Weijland [1990] is constructed as an initial algebra and is computable (43).
13. Several algebras in the list are made from the computable elements of uncountable algebras: the algebra  $CT^\infty(\Sigma, X)$  of all computable infinite terms over signature  $\Sigma$  in the set  $X$  of indeterminates (33); the algebras of recursive real numbers (37–39); and the algebra of computable infinite processes in the projective limit model of *ACP* (44). None of them are computable, but all of them can be arithmetically defined. Their effectiveness is considered in Sections 4 and 5 of this chapter.

We note that since every finite algebra is a computable algebra, the concepts of computable, semicomputable, and cosemicomputable algebras are *finiteness conditions* in algebra, i.e. isomorphism invariants possessed of all finite structures.

Let  $\Sigma$  be a signature with  $p$  constant symbols and  $q$  operations. For each computable  $\Sigma$ -algebra  $A$  there exists a recursive  $\Sigma$  algebra  $\Omega_\alpha$  of numbers and recursive congruence  $\equiv_\alpha$  which together form some machinery of the form

$$(\Omega_\alpha, \equiv_\alpha, c_1, \dots, c_p, f_1, \dots, f_q)$$

for computing in  $A$ . There are at most countably many such finite tuples of recursive sets, recursive relations, and recursive functions. Since at least one appropriate tuple of algorithms must be allocated to every isomorphism type of  $\Sigma$  algebras, the class of all isomorphism types of computable algebras is countably infinite. This argument also applies to the isomorphism types of semicomputable and cosemicomputable algebras, of course. However, many of the well-known algebraic theories have uncountably many countable algebras up to isomorphism.

**Theorem 2.2.7.** *Each collection of distinct isomorphism classes of countable abelian groups, rings, fields, and lattices is uncountable whereas each collection of isomorphism classes of computable abelian groups, rings, fields, and lattices is countable.*

Hence many other well-known classes of structures, such as groups, monoids, semigroups, integral domains, unique factorisation domains, and Euclidean domains share these cardinality properties. In fact there are many cardinality results that sharpen considerably our understanding of the computational and logical complexity of algebras. For example, among

the fruits of the embedding theory for finitely generated groups is the fact that

*there exist  $2^{\aleph_0}$  non-isomorphic two-generator groups*

(in Neumann [1937]) which can be sharpened by further embeddings to

*there exist  $2^{\aleph_0}$  non-isomorphic finitely generated simple groups*

(see Lyndon and Schupp [1977]).

Using arguments about cardinalities, we see that most algebraic systems are not computable.

Let us define the computable subsets and functions for an effectively numbered algebra.

**Definition 2.2.8 (Subsets).** Let  $A$  be an algebra effective under  $\alpha$ . Then a set  $S \subseteq A^k$  is  $\alpha$ -decidable,  $\alpha$ -semidecidable, or  $\alpha$ -cosemidecidable if its set

$$\alpha^{-1}(S) = \{(x_1, \dots, x_k) \in \Omega_\alpha^k : (\alpha(x_1), \dots, \alpha(x_k)) \in S\}$$

of numbers is recursive, r.e., or co-r.e., respectively.

Recalling item (19) in the list in 1.2, let  $F$  be a computable subfield of the complex numbers under  $\alpha$ . The set  $U(F)$  of all roots of unity in  $F$  is  $\alpha$ -semidecidable but need not be  $\alpha$ -decidable.

**Definition 2.2.9 (Mappings).** Let  $A$  and  $B$  be algebras of signature  $\Sigma$ , and let  $\phi : A \rightarrow B$  be any total function. Let  $\alpha : \Omega_\alpha \rightarrow A$  and  $\beta : \Omega_\beta \rightarrow B$  be effective numberings of  $A$  and  $B$ , respectively. Then  $\phi$  is an  $(\alpha, \beta)$ -computable map if there exists a recursive function  $f : \Omega_\alpha \rightarrow \Omega_\beta$  such that for all  $x \in \Omega_\alpha$ ,

$$\phi(\alpha(x)) = \beta(f(x));$$

or, equivalently, if  $f$  commutes the following diagram,

$$\begin{array}{ccc} A & \xrightarrow{\phi} & B \\ \alpha \uparrow & & \uparrow \beta \\ \Omega_\alpha & \xrightarrow{f} & \Omega_\beta \end{array}$$

Let  $Comp_{\alpha, \beta}(A, B)$  be the set of all  $(\alpha, \beta)$ -computable maps from  $A$  to  $B$ . An  $(\alpha, \beta)$ -computable homomorphism between  $A$  and  $B$  is a homomorphism that is an  $(\alpha, \beta)$ -computable map. Let  $Hom(A, B)$  be the set of all  $\Sigma$ -homomorphisms between  $A$  and  $B$ ; then set

$$Hom_{\alpha, \beta}(A, B) = Hom(A, B) \cap Comp_{\alpha, \beta}(A, B).$$

The following fact is easy to prove.

**Lemma 2.2.10.** *Let  $A, B$ , and  $C$  be  $\Sigma$ -algebras effective under  $\alpha, \beta$  and  $\gamma$ , respectively. Then if  $\phi \in \text{Comp}_{\alpha, \beta}(A, B)$  and  $\psi \in \text{Comp}_{\beta, \gamma}(B, C)$  then  $\psi \circ \phi \in \text{Comp}_{\alpha, \gamma}(A, C)$ .*

In the case that  $A = B$  and  $\alpha = \beta$  we let  $\text{Comp}_{\alpha}(A)$  be the set of all  $\alpha$ -computable maps on  $A$ . By Lemma 2.2.10,  $\text{Comp}_{\alpha}(A)$  is a subsemigroup of the semigroup  $F(A)$  of all functions on  $A$ .

The computable endomorphisms are defined similarly: let  $\text{End}(A)$  be the set of all endomorphisms of  $A$ ; then, trivially,

$$\text{End}_{\alpha}(A) = \text{End}(A) \cap \text{Comp}_{\alpha}(A).$$

Again by Lemma 2.2.10,  $\text{End}_{\alpha}(A)$  is a subsemigroup of  $\text{End}(A)$ .

Now, consider the bijective maps.

**Lemma 2.2.11.** *Let  $A$  and  $B$  be  $\Sigma$ -algebras effective under  $\alpha$  and  $\beta$ , respectively, and let  $\phi : A \rightarrow B$  be a  $\Sigma$ -isomorphism. Suppose that  $\beta$  is semi-computable. If  $\phi \in \text{Comp}_{\alpha, \beta}(A, B)$  then its inverse  $\phi^{-1} \in \text{Comp}_{\beta, \alpha}(B, A)$ .*

**Proof.** Let  $\phi$  be  $(\alpha, \beta)$ -computable and tracked by  $f : \Omega_{\alpha} \rightarrow \Omega_{\beta}$ . Define  $g : \Omega_{\beta} \rightarrow \Omega_{\alpha}$  by

$$g(x) = \begin{cases} (\text{some } z \in \Omega_{\alpha})[f(z) \equiv_{\beta} x] & \text{if } x \in \Omega_{\beta} \\ 0 & \text{if } x \notin \Omega_{\beta}. \end{cases}$$

Then  $g$  is total recursive on  $\mathbb{N}$  and it is easy to check that  $g$  tracks  $\phi^{-1}$  with respect to  $\alpha$  and  $\beta$ . ■

Let  $A$  be a  $\Sigma$ -algebra effective under  $\alpha$ . An automorphism  $\phi : A \rightarrow A$  is an  $\alpha$ -computable automorphism if both  $\phi$  and  $\phi^{-1}$  are  $\alpha$ -computable. By Lemma 2.2.10, the set  $\text{CAut}_{\alpha}(A)$  of all  $\alpha$ -computable automorphisms is a subgroup of  $\text{Aut}(A)$ . If  $A$  is semicomputable under  $\alpha$  then, by Lemma 2.2.10,

$$\text{CAut}_{\alpha}(A) = \text{Aut}(A) \cap \text{Comp}_{\alpha}(A).$$

Let  $\Sigma$  be a signature and  $X$  a countable set of variables, say  $X = \{x_1, \dots, x_n\}$  or  $X = \{x_1, x_2, \dots\}$ . Let  $T(\Sigma, X)$  be the  $\Sigma$ -term algebra over  $X$ . Then  $T(\Sigma, X)$  is a computable algebra under any of its usual Gödel numberings  $\gamma : \Omega_{\gamma} \rightarrow T(\Sigma, X)$ . We will describe the computability properties of term algebras that we will need.

**Definition 2.2.12 (Standard numberings).** Let  $\gamma$  be a computable numbering for  $T(\Sigma, X)$ . We say that  $\gamma$  is a *standard computable numbering* if we can decide when a term is a variable and which variable it is, and we can compute the subterms of a term that is not a variable.

Let us illustrate how these conditions may be made more precise: there exists a recursive function

$$d : \mathbb{N} \times \Omega_\gamma \rightarrow \mathbb{N},$$

such that

$$\begin{aligned} &\text{if } d(0, z) = 0 \text{ then } \gamma(z) = x_i \text{ where } d(1, z) = i \\ &\text{if } d(0, z) = 1 \text{ and } 1 \leq i \leq p \text{ then } \gamma(z) = c_i \text{ where } d(1, z) = i \\ &\text{if } d(0, z) = i \text{ and } 2 \leq i \leq q + 1 \text{ then} \\ &\quad \gamma(z) = \sigma_{i-1}(\gamma(d(1, z)), \dots, \gamma(d(k(i-1), z))). \end{aligned}$$

If  $\gamma$  is standard then it is possible to compute the length of terms.

Later, we shall see that all standard numberings are equivalent as computable numberings. Further assumptions on standard numberings of terms that are natural to make are: let  $\Omega_\gamma = \mathbb{N}$ , and let  $\gamma$  be a bijection, i.e.

$$\gamma : \Omega_\gamma \rightarrow T(\Sigma, X) \text{ and } \gamma^{-1} : T(\Sigma, X) \rightarrow \Omega_\gamma$$

be  $\Sigma$ -isomorphisms.

Let  $\gamma$  satisfy the following monotonic property: for each term  $t(x) \in T(\Sigma, X)$  with single variable  $x$ , then for all  $r_1, r_2 \in T(\Sigma, X)$ ,

$$\gamma^{-1}(r_1) < \gamma^{-1}(r_2) \text{ implies } \gamma^{-1}(t(r_1)) < \gamma^{-1}(t(r_2)).$$

This condition means that the numbering  $\gamma$  induces an ordering on terms and that substitution into terms is monotonic with respect to this ordering.

Consider the term evaluation function

$$TE : T(\Sigma, X) \times A^X \rightarrow A$$

for terms over  $T(\Sigma, X)$  given in Definition 2.1.7. In computing with this map  $TE$  we must consider the various cases for  $X$ .

Let  $X = \{x_1, \dots, x_n\}$ . Then  $A^X$  is represented by  $A^n$  and  $TE$  can be rewritten as

$$TE_n : T(\Sigma, X) \times A^n \rightarrow A.$$

The function is computable with respect to any standard numbering  $\gamma$  of  $T(\Sigma, X)$  for it is easy to define a tracking function

$$te_n : \Omega_\gamma \times \Omega_\alpha^n \rightarrow \Omega_\alpha$$

for  $TE_n$ . Now, let  $X = \{x_1, x_2, \dots\}$ . Then  $A^X$  is represented by the set  $A^*$  of finite sequences over  $A$ , and  $TE$  can be rewritten as

$$TE_\infty : T(\Sigma, X) \times A^* \rightarrow A.$$

The function is computable, tracked by some recursive function

$$te_\infty : \Omega_\gamma \times \Omega_\alpha^* \rightarrow \Omega_\alpha.$$

Let  $A$  be any countable algebra of signature  $\Sigma$ . Let  $a = \{a_i : i \in I\}$  be any set of generators for the algebra  $A$  (for example, an enumeration of all the elements of  $A$ ). Let  $X = \{x_i : i \in I\}$  be a corresponding list of indeterminates and consider the term evaluation map

$$\nu(a) : T(\Sigma, X) \rightarrow A$$

that substitutes  $a_i$  for  $x_i$  in the terms of the algebra  $T(\Sigma, X)$ . Because  $T(\Sigma, X)$  is free for the class of all  $\Sigma$ -algebras, the map  $\nu(a)$  is the unique homomorphic extension of the assignment map  $a(x_i) = a_i$ . Because  $a = \{a_i : i \in I\}$  generates  $A$ , the map  $\nu(a)$  is an epimorphism. We define

$$\gamma_a = \nu(a) \circ \gamma.$$

**Theorem 2.2.13.**  *$A$  has an effective numbering  $\gamma_a : \Omega_\gamma \rightarrow A$ .*

**Proof.** The function

$$\gamma_a : \Omega_\gamma \rightarrow A$$

is a  $\Sigma$ -epimorphism formed by composing the  $\Sigma$ -epimorphisms  $\gamma : \Omega_\gamma \rightarrow T(\Sigma, X)$  and  $\nu(a) : T(\Sigma, X) \rightarrow A$ . The recursive number algebra of the numbering is the domain of the function. Thus,  $\gamma_a$  is an effective numbering of  $A$ . ■

Given the epimorphism  $\nu(a) : T(\Sigma, X) \rightarrow A$  we can construct the natural isomorphism

$$A \cong T(\Sigma, X) / \equiv_{\nu(a)}$$

by means of the Homomorphism Theorem. Clearly, it is the case that for  $x, y \in \Omega_\gamma$ ,

$$x \equiv_{\gamma_a} y \Leftrightarrow \gamma(x) \equiv_{\nu(a)} \gamma(y).$$

Let us assume that the algebra  $A$  is  $\Sigma$ -minimal for a non-void signature  $\Sigma$ . Then we can rewrite  $\nu(a)$  as  $\nu_A$  and the above isomorphism as

$$A \cong T(\Sigma) / \equiv_{\nu_A}$$

where the closed term evaluation map  $\nu_A$  is uniquely defined and hence the congruence  $\equiv_{\nu_A}$  is unique (recall Lemma 2.1.9). More suggestively, we write the congruence  $\equiv_{\nu_A}$  as  $\equiv_A$ .



**Definition 2.2.14 (Word problem).** The algorithmic question

*given terms  $t_1$  and  $t_2$  from  $T(\Sigma)$ , is  $t_1 \equiv_A t_2$ ?*

is called the *word problem* for the algebra  $A$ ; it is named after the more specific problem of deciding the equality of words or terms over the signatures of semigroups or groups from finite presentations (see Example 3.2.9 below).

Let  $\gamma_A : \Omega_\gamma \rightarrow A$  be the effective numbering for  $\Sigma$ -minimal algebra  $A$  derived from  $\gamma$  and  $\nu_A$ . Now, for  $x, y \in \Omega_\gamma$ ,

$$x \equiv_{\gamma_A} y \Leftrightarrow \gamma(x) \equiv_A \gamma(y).$$

Thus,  $\equiv_A$  is  $\gamma$ -decidable,  $\gamma$ -semidecidable, or  $\gamma$ -cosemidecidable on  $T(\Sigma)$  if, and only if,  $A$  is computable, semicomputable, or cosemicomputable under  $\gamma_A$ , respectively. A stronger statement is true, namely:  $A$  is computable, semicomputable, or cosemicomputable if, and only if,  $\equiv_A$  is  $\gamma$ -decidable,  $\gamma$ -semidecidable, or  $\gamma$ -cosemidecidable, respectively. This characterisation of the decidability of the word problem requires results on the invariance of algorithms across sets of possible numberings.

Thus, the recursion-theoretic complexity of any algebra can be measured and classified by the complexity of its general word problem. As we will see terms and term algebras have an essential role in the theory of computation on abstract algebras.

## 2.3 Invariance

An algebra  $A$  is computable if *there exists* some computable numbering  $\alpha$  for  $A$ . For any computable algebra, there are many computable numberings and they may have desirable or undesirable properties. Shortly we will see that every computable algebra has a bijective numbering with code set  $\mathbb{N}$ . Let  $C(A)$  be the set of all computable numberings of the algebra  $A$ . The *choice* of a numbering  $\alpha \in C(A)$  suggests that the effectiveness of a subset or function on  $A$  may vary over the set  $C(A)$  of possible computable numberings. To illustrate, let  $S \subseteq A$  and consider the following questions:

*Is  $S$  decidable in all computable numberings of  $A$ ?*

*Is  $S$  decidable in some computable numbering of  $A$ , but undecidable in others?*

*Is  $S$  undecidable in all numberings of  $A$ ?*

Another question concerns the invariance of computable maps. If  $A$  is computable under two numberings  $\alpha$  and  $\beta$  then what is the relation between the two sets  $Comp_\alpha(A)$  and  $Comp_\beta(A)$  of computable functions on  $A$ ? What is

$$Comp(A) = \bigcap_{\alpha \in C(A)} Comp_\alpha(A)?$$

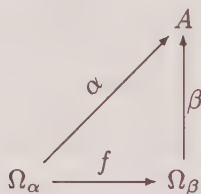
More generally, what properties of what algebras are decidable or undecidable in all computable numberings?

These constitute a small fraction of the number of questions that concern numberings. The first step in understanding invariance is to define the basic idea of the equivalence of two effective numberings; then we will focus on the case when these numberings are computable and semicomputable numberings.

**Definition 2.3.1 (Reductions between effective numberings).** Let  $A$  be an algebra with effective numberings  $\alpha : \Omega_\alpha \rightarrow A$  and  $\beta : \Omega_\beta \rightarrow A$ . Then  $\alpha$  *recursively reduces* to  $\beta$  if there exists a recursive function  $f : \Omega_\alpha \rightarrow \Omega_\beta$  such that for all  $x \in \Omega_\alpha$ ,

$$\alpha(x) = \beta f(x);$$

or, equivalently, if  $f$  commutes the diagram:



We call  $f$  a *reduction map*. The idea is that for any  $\alpha$ -code for an element of  $A$  we can compute some  $\beta$ -code for that element. We write  $\alpha \leq \beta$ .

We will show that the decidability of subsets of an algebra  $A$  (as in Definition 2.2.8) can be transferred under a reduction of numberings. First, we recall the definition of many-one reducibility. Let  $X \subseteq \mathbb{N}^p$  and  $Y \subseteq \mathbb{N}^q$ . We say that  $X$  is *many-one reducible* to  $Y$  if there exists a total recursive function  $f : \mathbb{N}^p \rightarrow \mathbb{N}^q$  such that for all  $x \in \mathbb{N}^p$ ,

$$x \in X \Leftrightarrow f(x) \in Y.$$

**Lemma 2.3.2.** *Let  $A$  be an algebra with effective numberings  $\alpha$  and  $\beta$ . Let  $S \subseteq A^k$  be any subset. If*

$$\alpha \leq \beta$$

*then  $\alpha^{-1}(S)$  is many-one reducible to  $\beta^{-1}(S)$ , and consequently:*

1. *if  $S$  is  $\beta$ -decidable then  $S$  is  $\alpha$ -decidable;*
2. *if  $S$  is  $\beta$ -semidecidable then  $S$  is  $\alpha$ -semidecidable;*
3. *if  $S$  is  $\beta$ -cosemidecidable then  $S$  is  $\alpha$ -cosemidecidable.*

*In particular, the word problem  $\equiv_\alpha$  of  $\alpha$  is many-one reducible to  $\equiv_\beta$  of  $\beta$  and hence:*

4. if  $\beta$  is a computable numbering of  $A$  then  $\alpha$  is also computable;
5. if  $\beta$  is a semicomputable numbering of  $A$  then  $\alpha$  is also semicomputable;
6. if  $\beta$  is a cosemicomputable numbering of  $A$  then  $\alpha$  is also cosemicomputable.

**Proof.** Let  $\alpha \leq \beta$  via the reduction map  $f : \Omega_\alpha \rightarrow \Omega_\beta$ . Then, for all  $x_1, \dots, x_k \in \mathbb{N}$ ,

$$\begin{aligned} (x_1, \dots, x_k) \in \alpha^{-1}(S) &\Leftrightarrow (\alpha(x_1), \dots, \alpha(x_k)) \in S \\ &\Leftrightarrow (\beta f(x_1), \dots, \beta f(x_k)) \in S \text{ since } \alpha = \beta f \\ &\Leftrightarrow (f(x_1), \dots, f(x_k)) \in \beta^{-1}(S). \end{aligned}$$

Thus,  $\alpha^{-1}(S)$  is many-one reducible to  $\beta^{-1}(S)$  by  $f$ ; more precisely, the many-one reduction is a map  $f_0 : \mathbb{N}^k \rightarrow \mathbb{N}^k$  constructed from  $f$  as follows:

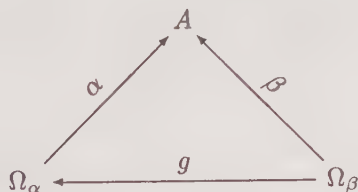
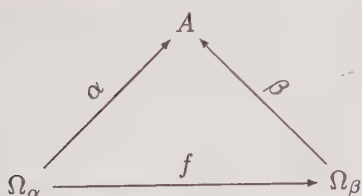
$$f_0(x_1, \dots, x_k) = \begin{cases} (f(x_1), \dots, f(x_k)) & \text{if } (x_1, \dots, x_k) \in \Omega_\alpha^k \\ (0, \dots, 0) & \text{if } (x_1, \dots, x_k) \notin \Omega_\alpha^k. \end{cases}$$

The other clauses (1)–(6) are immediate consequences. ■

**Definition 2.3.3 (Recursive equivalence of effective numberings).** Let  $A$  be an algebra with effective numberings  $\alpha : \Omega_\alpha \rightarrow A$  and  $\beta : \Omega_\beta \rightarrow A$ . Then  $\alpha$  is *recursively equivalent* to  $\beta$  if  $\alpha$  recursively reduces to  $\beta$  and  $\beta$  recursively reduces to  $\alpha$ . We write  $\alpha \sim \beta$  and note that

$$\alpha \sim \beta \text{ if, and only if, } \alpha \leq \beta \text{ and } \beta \leq \alpha.$$

Thus, if  $\alpha \sim \beta$  then there exist recursive functions  $f, g$  which commute the diagrams:



The idea is that the  $\alpha$ -codes for any element of  $A$  can be computably interchanged with the  $\beta$ -codes for that element. An immediate consequence of Lemma 2.3.2 is the following.

**Lemma 2.3.4.** Let  $A$  be an algebra with effective numberings  $\alpha$  and  $\beta$ . Let  $S \subseteq A^k$  be any subset. If

$$\alpha \sim \beta$$

then  $\alpha^{-1}(S)$  is many-one equivalent to  $\beta^{-1}(S)$  and consequently:

1.  $S$  is  $\beta$ -decidable if, and only if,  $S$  is  $\alpha$ -decidable;
2.  $S$  is  $\beta$ -semidecidable if, and only if,  $S$  is  $\alpha$ -semidecidable;
3.  $S$  is  $\beta$ -cosemidecidable if, and only if,  $S$  is  $\alpha$ -cosemidecidable.

In particular, the word problem  $\equiv_\alpha$  is many-one equivalent to  $\equiv_\beta$  and hence:

4.  $\beta$  is a computable numbering if, and only if,  $\alpha$  is a computable numbering;
5.  $\beta$  is a semicomputable numbering if, and only if,  $\alpha$  is a semicomputable numbering;
6.  $\beta$  is a cosemicomputable numbering if, and only if,  $\alpha$  is a cosemicomputable numbering.

The concept of recursive equivalence allows us to give a precise meaning to the idea of a standard Gödel numbering of terms as defined in Definition 2.2.12.

**Lemma 2.3.5.** *Let  $T(\Sigma, X)$  be any term algebra and  $\alpha$  and  $\beta$  be any computable numberings of  $T(\Sigma, X)$  that are standard. Then  $\alpha \sim \beta$ .*

**Proof.** It is easy to construct reduction maps using the decomposition functions  $d_\alpha$  and  $d_\beta$  of  $\alpha$  and  $\beta$ , respectively. ■

The computable functions are also uniquely defined up to recursive equivalence.

**Lemma 2.3.6.** *Let  $A$  be an algebra effective under both  $\alpha$  and  $\beta$ . If  $\alpha \sim \beta$  then*

$$\text{Comp}_\alpha(A) = \text{Comp}_\beta(A).$$

**Proof.** Let  $\alpha : \Omega_\alpha \rightarrow A$  and  $\beta : \Omega_\beta \rightarrow A$ , let  $\alpha \leq \beta$  by  $f$  and  $\beta \leq \alpha$  by  $g$ . Let  $\phi : A \rightarrow A$  be  $\alpha$ -computable with tracking function  $h$ . We construct a tracking function for  $\phi$  in  $\beta$ . Now, for  $x \in \Omega_\alpha$ ,

$$\begin{aligned} \phi\beta(x) &= \phi\alpha g(x) && \text{since } \beta = \alpha g; \\ &= \alpha h g(x) && \text{since } h \text{ tracks } \phi \text{ in } \alpha; \\ &= \beta f h g(x) && \text{since } \alpha = \beta f; \end{aligned}$$

hence  $\phi$  is tracked by the recursive function  $f h g : \Omega_\beta \rightarrow \Omega_\beta$ . ■

The following simple fact will prove to be influential in understanding the notions of computable and semicomputable algebras.

**Lemma 2.3.7 (Equivalence of semicomputable and computable numberings).** *Let  $A$  be an algebra with effective numberings  $\alpha$  and  $\beta$ . Suppose that  $\beta$  is semicomputable. If  $\alpha$  is recursively reducible to  $\beta$  then*

$\beta$  is recursively reducible to  $\alpha$  and, hence,  $\alpha$  and  $\beta$  are recursively equivalent, and  $\alpha$  is semicomputable. Moreover, if  $\beta$  is computable then  $\alpha$  is computable.

**Proof.** Let  $\alpha$  recursively reduce to  $\beta$  by means of a total recursive function  $f : \Omega_\alpha \rightarrow \Omega_\beta$ . Thus, for all  $x \in \Omega_\alpha$ ,  $\alpha(x) = \beta f(x)$ . Define  $g : \Omega_\beta \rightarrow \Omega_\alpha$  by

$$g(y) = (\text{some } z \in \Omega_\alpha)[f(z) \equiv_\beta y].$$

To show that for  $y \in \Omega_\beta$ ,  $\beta(y) = \alpha g(y)$ , we calculate as follows: by definition of  $g$ , for some  $z$  such that  $f(z) \equiv_\beta y$  we have

$$\begin{aligned} \alpha g(y) &= \alpha(z); \\ &= \beta f(z) \quad \text{since } \alpha \leq \beta \text{ via } f; \\ &= \beta(y). \end{aligned}$$

Thus,  $\beta \leq \alpha$  via  $g$ .

That  $\beta$  is semicomputable or computable implies that  $\alpha$  is semicomputable or computable, respectively, follows from Lemma 2.3.4. ■

We examine some simple types of special numberings for algebras.

**Lemma 2.3.8.** *Let  $A$  be an infinite countable algebra. For every effective numbering  $\alpha$  there exists an effective numbering  $\beta$  with code set  $\mathbb{N}$  such that  $\beta \leq \alpha$ .*

**Proof.** Let  $A$  be effective under  $\alpha : \Omega_\alpha \rightarrow A$ . Since  $A$  is infinite and  $\alpha$  is an epimorphism, the set  $\Omega_\alpha$  of numbers is infinite. Let  $e : \mathbb{N} \rightarrow \Omega_\alpha$  be a recursive function enumerating the code set  $\Omega_\alpha$  without repetitions. Using composition, define the new map

$$\beta = \alpha \circ e : \mathbb{N} \rightarrow A.$$

Clearly, the map is surjective.

We equip  $\mathbb{N}$  with tracking functions as follows: let  $f : \Omega_\alpha^k \rightarrow \Omega_\alpha$  be the tracking function of the  $k$ -ary operation  $\sigma_A$  of  $A$  and define  $g : \mathbb{N}^k \rightarrow \mathbb{N}$  by

$$g(x_1, \dots, x_k) = e^{-1}(f(e(x_1), \dots, e(x_k))).$$

We check that  $g$  is a tracking function for  $\sigma_A$  in  $\beta$ :



$$\begin{aligned}
\sigma_A(\beta(x_1), \dots, \beta(x_k)) &= \sigma_A(\alpha(e(x_1)), \dots, \alpha(e(x_k))) \\
&\quad \text{by definition of } \beta; \\
&= \alpha(f(e(x_1), \dots, e(x_k))) \\
&\quad f \text{ tracks } \sigma_A \text{ in } \alpha; \\
&= \alpha e e^{-1}(f(e(x_1), \dots, e(x_k))) \\
&= (\alpha e)(e^{-1}(f(e(x_1), \dots, e(x_k)))) \\
&= (\alpha e)(g(x_1, \dots, x_k)) \\
&\quad \text{by definition of } g; \\
&= \beta(g(x_1, \dots, x_k)). \\
&\quad \text{by definition of } \beta.
\end{aligned}$$

Thus,  $\beta$  is an epimorphism and an effective numbering with code set  $\mathbb{N}$ .

We note that by definition  $\beta \leq \alpha$  by the map  $e$ ; so for example:

$$\begin{aligned}
x \equiv_\beta y &\Leftrightarrow \alpha e(x) = \alpha e(y) \text{ by definition of } \beta \\
&\Leftrightarrow e(x) \equiv_\alpha e(y).
\end{aligned}$$

■

We now prove a stronger result that holds only for computable algebras.

**Lemma 2.3.9 (Representation Lemma).** *Let  $A$  be an infinite algebra. For every computable numbering  $\alpha$  of  $A$  there exists a bijective computable numbering  $\beta$  with code set  $\mathbb{N}$  such that  $\beta \sim \alpha$ . Thus, every computable algebra  $A$  is isomorphic to a recursive number algebra  $R$  whose carrier is the set  $\mathbb{N}$  of natural numbers.*

*If  $A$  is finite of cardinality  $m$  then  $A$  is isomorphic with a recursive number algebra whose carrier set is the set  $\{0, 1, \dots, m-1\}$  of the first  $m$  natural numbers.*

**Proof.** The argument is an adaptation of that for Lemma 2.3.8. Let  $A$  be computable under  $\alpha : \Omega_\alpha \rightarrow A$ . Since  $A$  is infinite and  $\alpha$  is an epimorphism, the set  $\Omega_\alpha$  of numbers is infinite. Define a subset  $T_\alpha$  of  $\Omega_\alpha$  from  $\Omega_\alpha / \equiv_\alpha$  by using

$$n(x) = (\text{least } z)[z \equiv_\alpha x]$$

to calculate the smallest  $\alpha$ -number for an element of  $A$  and set  $T_\alpha = \text{im}(n)$ . The set  $T_\alpha$  contains one and only one  $\alpha$ -code for each element of  $A$  and is called a *transversal*. The set is infinite. Let  $e : \mathbb{N} \rightarrow T_\alpha$  be a recursive function enumerating the transversal  $T_\alpha$  without repetitions. Using composition, define a new map

$$\beta = \alpha \circ e : \mathbb{N} \rightarrow A.$$

It is easy to check that the map is surjective. The tracking functions are defined as before: let  $f : \Omega_\alpha^k \rightarrow \Omega_\alpha$  be the tracking function for the  $k$ -ary operation  $\sigma_A$  of  $A$  and define  $g : \mathbb{N}^k \rightarrow \mathbb{N}$  by

$$g(x_1, \dots, x_k) = e^{-1} f(e(x_1), \dots, e(x_k)).$$

Clearly,

$$\begin{aligned} x \equiv_{\beta} y &\Leftrightarrow \alpha e(x) = \alpha e(y) \text{ in } A \\ &\Leftrightarrow e(x) \equiv_{\alpha} e(y) \end{aligned}$$

and is recursive. Thus,  $\beta$  is a bijective computable numbering with code set  $N$ .

Now  $\beta \leq \alpha$  by  $e$  and, since  $\alpha$  is computable,  $\alpha \leq \beta$  by Lemma 2.3.7. The case of finite algebras we leave as an exercise. ■

For example, any computable numbering of  $T(\Sigma, X)$  is equivalent to a bijective computable numbering with code set  $N$ .

Obviously, no such isomorphic representation is possible for the semi-computable algebras for otherwise they would be computable.

Next we turn to the finitely generated algebras; these enjoy special invariance properties.

Let  $A$  be any finitely generated algebra. Without loss of generality we may suppose that  $A$  is minimal with respect to signature  $\Sigma$ . This means that some finite set of generators has been chosen and added to the signature. Let  $\gamma_A$  be any standard effective numbering of  $A$  that has been derived from any standard numbering  $\gamma$  of  $T(\Sigma)$  via the unique evaluation mapping  $\nu_A : T(\Sigma) \rightarrow A$ .

**Theorem 2.3.10 (Basic Reduction Theorem).** *Let  $A$  be any finitely generated algebra. Let  $\alpha$  be any effective numbering of  $A$ . Then the standard effective numbering  $\gamma_A : \Omega_{\gamma} \rightarrow A$  is recursively reducible to  $\alpha$ , i.e.*

$$\gamma_A \leq \alpha.$$

**Proof.** Let  $\alpha : \Omega_{\alpha} \rightarrow A$  and let  $c_{\alpha}$  be a code for constant  $c_A$  and let  $\sigma_{\alpha}$  be a tracking function for operation  $\sigma_A$ . We define a recursive function  $f : \Omega_{\gamma} \rightarrow \Omega_{\alpha}$  that reduces  $\gamma_A$  to  $\alpha$ . This is done by a course-of-values recursion based on codes for the terms in  $T(\Sigma)$ . By Lemma 2.3.9, we may assume that  $\gamma$  is bijective. Let  $t \in T(\Sigma)$  and let  $\langle t \rangle$  denote the unique  $\gamma$ -code for  $t$  in  $\Omega_{\gamma}$ , i.e.  $\gamma(\langle t \rangle) = t$ .

We define the function as follows:

$$\begin{aligned} f(\langle c \rangle) &= c_{\alpha} \\ f(\langle \sigma(t_1, \dots, t_k) \rangle) &= \sigma_{\alpha}(f(\langle t_1 \rangle), \dots, f(\langle t_k \rangle)) \end{aligned}$$

and

$$f(x) = 0 \quad \text{if } x \notin \Omega_{\gamma}.$$

**Claim.** For  $x \in \Omega_{\gamma}$ ,  $\gamma_A(x) = \alpha f(x)$ .

This is proved by induction on the codes for terms. The basis case is that of codes for constants:

$$\alpha f(\langle c \rangle) = \alpha(c_\alpha) = c_A = \gamma_A(\langle c \rangle).$$

Suppose the result is true for all codes for all terms of lower complexity than  $t = \sigma(t_1, \dots, t_k)$ . Then

$$\alpha f(\langle \sigma(t_1, \dots, t_k) \rangle) = \alpha(\sigma_\alpha(f(\langle t_1 \rangle), \dots, f(\langle t_k \rangle)))$$

by definition of  $f$ ;

$$= \sigma_A(\alpha(f(\langle t_1 \rangle)), \dots, \alpha(f(\langle t_k \rangle)))$$

since  $\alpha$  is a homomorphism;

$$= \sigma_A(\gamma_A(\langle t_1 \rangle), \dots, \gamma_A(\langle t_k \rangle))$$

by the induction hypothesis;

$$= \gamma_A(\langle \sigma(t_1, \dots, t_k) \rangle)$$

since  $\gamma_A$  is a homomorphism. ■

**Corollary 2.3.11.** *Let  $A$  be a finitely generated algebra with effective numbering  $\alpha$ . Let  $S \subseteq A^k$ . Then  $\gamma_A^{-1}(S)$  is many-one reducible to  $\alpha^{-1}(S)$ . Thus,  $A$  is computable, semicomputable, or cosemicomputable if, and only if, the word problem  $\equiv_A$  for  $A$  is  $\gamma$ -decidable,  $\gamma$ -semidecidable, or  $\gamma$ -cosemidecidable, respectively.*

Let  $E(A)$  be the set of all effective numberings of  $A$ . Then  $\leq$  induces a partial ordering of  $E(A)$  and  $\sim$  is an equivalence relation on  $E(A)$ . We call the factor structure  $E(A)/\sim$  the *effective spectrum* of  $A$ ; this set can be ordered by  $\leq$ . The structure of  $E(A)$  can be very complicated as the structure of the set of numberings of the partial recursive functions amply demonstrates (see Rogers [1967] and Mal'cev [1970]). We can apply the same observations to the sets  $SC(A)$  and  $C(A)$  of semicomputable and computable numberings. By Lemma 2.3.7, we know that the *semicomputable spectrum*  $SC(A)/\sim$  and the *computable spectrum*  $C(A)/\sim$  are discrete in their induced orderings.

**Definition 2.3.12 (Stability).** An algebra  $A$  is said to be *computably stable* if any two computable numberings of  $A$  are recursively equivalent; equivalently, if  $|C(A)/\sim| = 1$ .

An algebra  $A$  is said to be *semicomputably stable* if any two semicomputable numberings of  $A$  are recursively equivalent; equivalently, if  $|SC(A)/\sim| = 1$ .

Clearly, semicomputable stability implies computable stability. The following fact is straightforward to check.

**Lemma 2.3.13.** *Semicomputable stability and computable stability are finiteness conditions.*

The main invariance property of interest is the following result from Mal'cev [1961]:

**Theorem 2.3.14 (Invariance Theorem).** *Let  $A$  be a finitely generated algebra. If  $A$  is semicomputable under both  $\alpha$  and  $\beta$  then  $\alpha$  and  $\beta$  are recursively equivalent, i.e.  $A$  is semicomputably stable and hence computably stable.*

**Proof.** If  $A$  is finitely generated then we may assume it is  $\Sigma$ -minimal for some finite non-void signature  $\Sigma$ . Let  $\gamma_A$  be a standard effective numbering of  $A$  built from  $T(\Sigma)$ . By Lemma 2.3.10, if  $\alpha : \Omega_\alpha \rightarrow A$  is a semicomputable numbering of  $A$  then  $\gamma_A \leq \alpha$  and  $\gamma_A$  is a semicomputable numbering of  $A$ . By Lemma 2.3.7,  $\alpha \leq \gamma_A$  and the numberings are recursively equivalent. Thus, all semicomputable numberings are equivalent to the standard numberings and hence to one another. ■

Computable stability is not confined to finitely generated algebras. In Mal'cev [1971, Chapter 22] there are the following examples. Let  $F$  be a field that is a finite algebraic extension of the field  $\mathbb{Q}$  of rational numbers. Then  $F$  is computable and computably stable. Further examples are the special linear group  $SL(n, F)$  and its subgroup  $RSL(n, F)$  of triangular matrices over  $F$ .

**Lemma 2.3.15.** *Let  $A$  be computable under  $\alpha$ . For any  $\phi \in \text{Aut}(A)$ ,  $\phi\alpha$  is a computable numbering and*

$$\alpha \sim \phi\alpha \text{ if, and only if, } \phi \in C\text{Aut}_\alpha(A).$$

*If  $A$  is computably stable then*

$$\text{Aut}(A) = C\text{Aut}(A) = C\text{Aut}_\alpha(A).$$

*If there is a computable numbering  $\alpha$  such that*

$$C\text{Aut}_\alpha(A) \neq \text{Aut}(A)$$

*then  $A$  is not computably stable.*

The last condition provides many examples of algebras that are not computably stable because of the following consequence:

**Lemma 2.3.16.** *Let  $A$  be a computable algebra with uncountably many automorphisms. Then  $A$  is not computably stable and in fact the computable spectrum  $C(A)/\sim$  is uncountable.*

The simplest example of a computable algebra with uncountably many automorphisms is a term algebra  $T(\Sigma, X)$  with  $X = \{x_1, x_2, \dots\}$ . This is because any bijective map  $\phi_0 : X \rightarrow X$  extends uniquely to an automorphism  $\phi : T(\Sigma, X) \rightarrow T(\Sigma, X)$ .

The structure of the class of numberings is an important topic of study. The following remarkable theorem is due to S. Goncharov:

**Theorem 2.3.17 (Goncharov's Theorem).** *For each  $n \geq 1$  there exists a computable algebra  $A$  such that  $|C(A)/\sim| = n$ . In detail: there exist  $n$  computable numberings  $\alpha_1, \dots, \alpha_n$  of  $A$  such that*

1. *for each  $1 \leq i, j \leq n, \alpha_i \sim \alpha_j \Leftrightarrow i = j$ ; and*
2. *for any computable numbering  $\alpha$ ,*

$$\alpha \sim \alpha_1 \text{ or } \dots \text{ or } \alpha \sim \alpha_n.$$

In Goncharov [1980a; 1980b] there are given familiar examples of algebras satisfying the above, such as metabelian groups.

Finally, we note that there is a second important method of classifying effective numberings based on transformations of an algebra by automorphisms.

**Definition 2.3.18.** A set  $S \subseteq A^k$  is an *automorphism invariant* of  $A$  if for all  $(a_1, \dots, a_k) \in A^k$  and any  $\phi \in \text{Aut}(A)$ ,

$$(a_1, \dots, a_k) \in S \text{ if, and only if, } (\phi(a_1), \dots, \phi(a_k)) \in S.$$

For example, any subset of  $A$  defined by a first-order formula, or as the halting set of a while program, is an automorphism invariant.

**Definition 2.3.19 (Autoequivalence).** Let  $A$  be an algebra with effective numberings  $\alpha$  and  $\beta$ . Then  $\alpha$  is *recursively autoequivalent* to  $\beta$  if there exists some automorphism  $\phi \in \text{Aut}(A)$  such that the effective numbering  $\phi\alpha$  is recursively equivalent to  $\beta$ . We write  $\alpha \approx \beta$ .

**Lemma 2.3.20.** *Let  $A$  be an algebra with effective numberings  $\alpha$  and  $\beta$ . Let  $S \subseteq A^k$  be an automorphism invariant. If  $\alpha \approx \beta$  then  $\alpha^{-1}(S)$  is many-one equivalent to  $\beta^{-1}(S)$  and consequently:*

1.  *$S$  is  $\beta$ -decidable if, and only if,  $S$  is  $\alpha$ -decidable;*
2.  *$S$  is  $\beta$ -semidecidable if, and only if,  $S$  is  $\alpha$ -semidecidable;*
3.  *$S$  is  $\beta$ -cosemidecidable if, and only if,  $S$  is  $\alpha$ -cosemidecidable.*

*In particular, the word problem  $\equiv_\alpha$  is many-one equivalent to  $\equiv_\beta$  and hence:*

4.  *$\beta$  is a computable numbering if, and only if,  $\alpha$  is a computable numbering;*



5.  $\beta$  is a semicomputable numbering if, and only if,  $\alpha$  is a semicomputable numbering;
6.  $\beta$  is a cosemicomputable numbering if, and only if,  $\alpha$  is a cosemicomputable numbering.

**Proof.** Let  $\phi \in \text{Aut}(A)$ . Then, for all  $x_1, \dots, x_k \in \mathbb{N}$ ,

$$\begin{aligned}
 (x_1, \dots, x_k) \in \alpha^{-1}(S) &\Leftrightarrow (\alpha(x_1), \dots, \alpha(x_k)) \in S \\
 &\Leftrightarrow (\phi\alpha(x_1), \dots, \phi\alpha(x_k)) \in S \text{ since } S \text{ is invariant} \\
 &\Leftrightarrow (x_1, \dots, x_k) \in (\phi\alpha)^{-1}(S).
 \end{aligned}$$

By Lemma 2.3.4, many-one equivalence and clauses (1)–(6) follow from the fact that  $\phi\alpha \sim \beta$  for some  $\phi \in \text{Aut}(A)$ . ■

**Definition 2.3.21 (Autostability).** An algebra  $A$  is said to be *computably autostable* if any two computable numberings are recursively autoequivalent; equivalently, if  $|C(A)/\approx| = 1$ .

An algebra  $A$  is said to be *semicomputably autostable* if any two semicomputable numberings are recursively autoequivalent; equivalently, if  $|SC(A)/\approx| = 1$ .

Clearly, semicomputable autostability implies computable autostability.

**Lemma 2.3.22.** *Semicomputable autostability and computable autostability are finiteness conditions.*

For example, the field  $\mathbb{Q}(\sqrt{p_1}, \sqrt{p_2}, \dots)$  is computably autostable. Examples of autostable abelian groups are discussed in Mal'cev [1971, Chapter 24].

Since the structure provided for Theorem 2.3.17 is rigid (i.e.  $\text{Aut}(A) = 1$ ) the result about finitely many numberings up to recursive equivalence implies the corresponding result about finitely many numberings up to autoequivalence.

## 2.4 A few computable constructions

We will describe a few elementary constructions of computable algebras in order to round off the basic definitions. We will concentrate on computable algebras and leave the reader to consider how the ideas can be adapted to semicomputable algebras. Our first question is:

*What are the computable subalgebras of a computable algebra?*

**Definition 2.4.1.** Let  $A$  be a  $\Sigma$ -algebra computable under  $\alpha$ . Let  $B$  be a  $\Sigma$ -subalgebra of  $A$ . Then  $B$  is an  $\alpha$ -semidecidable subalgebra of  $A$  if  $B$  is an  $\alpha$ -semidecidable subset of  $A$ , i.e.

$$\alpha^{-1}(B) = \{x \in \Omega_\alpha \mid \alpha(x) \in B\}$$

is r.e. An alternate name that we will use for this concept is  $\alpha$ -computable subalgebra. Let us see why the term  $\alpha$ -computable subalgebra is appropriate to our needs:

**Lemma 2.4.2.** *Let  $A$  be an algebra computable under  $\alpha$  and let  $B$  be an  $\alpha$ -computable subalgebra of  $A$ . Then  $B$  has a computable numbering  $\beta$  for which the inclusion map  $i : B \rightarrow A$  is  $(\beta, \alpha)$ -computable.*

**Proof.** The argument is similar to that of Lemma 2.3.8. Since  $\alpha^{-1}(B)$  is r.e. there is a recursive bijection  $e : \Omega_\beta \rightarrow \alpha^{-1}(B)$  that enumerates the set without repetition where  $\Omega_\beta$  is recursive. Define  $\beta = \alpha \circ e : \Omega_\beta \rightarrow B$ . Now, we can see that  $i : B \rightarrow A$  is  $(\beta, \alpha)$ -computable with tracking function  $e$  because for  $x \in \Omega_\beta$ ,

$$i(\beta(x)) = \alpha e(x),$$

by definition. We show that  $\beta$  is computable.

We equip  $\Omega_\beta$  with tracking functions using the inverse  $e^{-1}$  which is partial recursive, but total on  $\alpha^{-1}(B)$ . Let  $f$  be the tracking function for an operation  $\sigma_B$  of  $B$  with respect to  $\alpha$ . Then a tracking function  $g$  for  $\sigma_B$  with respect to  $\beta$  is defined by

$$g(x_1, \dots, x_k) = e^{-1}f(e(x_1), \dots, e(x_k)),$$

which is easy to check. To see that  $\equiv_\beta$  is recursive we observe that for any  $x, y \in \Omega_\beta$ ,

$$\begin{aligned} x \equiv_\beta y &\Leftrightarrow \alpha e(x) = \alpha e(y) \text{ in } B \\ &\Leftrightarrow e(x) \equiv_\beta e(y). \end{aligned}$$

**Lemma 2.4.3.** *Let  $A$  be an algebra computable under  $\alpha$ . The following are equivalent:*

1.  $B$  is an  $\alpha$ -computable subalgebra of  $A$ ;
2.  $B$  is the subalgebra  $\langle H \rangle$  of  $A$  generated by an  $\alpha$ -semidecidable subset  $H$  of  $A$ .

**Proof.** The fact that (1) implies (2) is obvious since we may take  $H = B$ . To see that (2) implies (1) we take  $\Omega = \alpha^{-1}(H)$  in the following lemma.

**Lemma 2.4.4.** *Let  $A$  be an algebra computable under  $\alpha : \Omega_\alpha \rightarrow A$ . Let  $\Omega$  be an r.e. subset of  $\Omega_\alpha$ . Then the subalgebra  $\langle \alpha(\Omega) \rangle$  of  $A$  generated by  $\alpha(\Omega)$  is an  $\alpha$ -computable subalgebra of  $A$ .*

**Proof.** Recall that the term evaluation function

$$TE_\infty : T(\Sigma, X) \times A^* \rightarrow A$$

for terms over a countably infinite set  $X$  of variables is computable, tracked by some recursive function

$$te_\infty : \Omega_\gamma \times \Omega_\alpha^* \rightarrow \Omega_\alpha,$$

for any standard numbering  $\gamma$  of the term algebra. Thus,

$$x \in \alpha^{-1}(\langle \alpha(\Omega) \rangle) \Leftrightarrow (\exists z \in \Omega_\gamma)(\exists y_1, \dots, y_n \in \Omega_\alpha)[x \equiv_\alpha te_\infty(z, y_1, \dots, y_n)]$$

which is r.e. ■

It is now easy to deduce the following invaluable fact:

**Corollary 2.4.5.** *Every finitely generated subalgebra of a computable algebra is a computable subalgebra.*

The notion of computable (or semidecidable) subalgebra is complemented by the following:

**Definition 2.4.6.** Let  $A$  be an algebra computable under  $\alpha$ . Let  $B$  be a subalgebra of  $A$ . Then  $B$  is an  $\alpha$ -decidable subalgebra of  $A$  if  $B$  is an  $\alpha$ -decidable subset of  $A$ , i.e.

$$\alpha^{-1}(B) = \{x \in \Omega_\alpha \mid \alpha(x) \in B\}$$

is recursive.

**Examples 2.4.7.** A standard question that arises when working with a computable algebra  $A$  with computable numbering  $\alpha$  is

*Given any  $a_1, \dots, a_n \in A$ , is the subalgebra  $\langle a_1, \dots, a_n \rangle$  generated by  $a_1, \dots, a_n$  an  $\alpha$ -decidable subalgebra?*

Very commonly the answer is: no. For example, consider the problem in groups of matrices. Let  $\mathbb{Z}$  be the ring of integers and  $M(n, \mathbb{Z})$  be the ring of  $n \times n$  matrices over  $\mathbb{Z}$ . Consider the group  $G(n, \mathbb{Z})$  of all invertible matrices over  $\mathbb{Z}$  with integer coefficients; in fact,

$$GL(n, \mathbb{Z}) = \{m \in M(n, \mathbb{Z}) \mid \det(m) = 1 \text{ or } \det(m) = -1\}.$$

Clearly,  $GL(n, \mathbb{Z})$  is a computable group; indeed, it is computably stable. Every finitely generated subgroup of  $GL(n, \mathbb{Z})$  is a computable subgroup, i.e. is a semidecidable subset. C. F. Miller III has shown that

*For  $n \geq 4$ , there exists a finitely generated subgroup  $L$  of  $GL(n, \mathbb{Z})$  which is semidecidable but not decidable.*

In fact, the group  $L$  has an undecidable conjugacy problem and, for every recursively enumerable Turing degree, there is a finitely generated subgroup such that the membership problem has that degree. The group  $L$  can be chosen to be a subgroup of

$$SL(n, \mathbb{Z}) = \{m \in GL(n, \mathbb{Z}) \mid \det(m) = 1\}$$

which is a subgroup of  $GL(n, \mathbb{Z})$ . See Miller [1971] for these and other related results.

Let us consider the situation in rings. Every subring of the ring  $\mathbb{Z}$  is  $\alpha$ -decidable in every computable numbering  $\alpha$  of  $\mathbb{Z}$ . By the Hilbert Basis Theorem, every ideal of  $\mathbb{Z}[X_1, \dots, X_k]$  is finitely generated. Indeed:

*Every ideal of  $\mathbb{Z}[X_1, \dots, X_k]$  is decidable in every computable numbering of the polynomial ring.*

This fact follows from important work of Greta Hermann in 1926, on constructive aspects of polynomial rings (see Hermann [1926]).

Now we turn to homomorphisms and congruences to establish a computable version of the Homomorphism Theorem.

**Lemma 2.4.8.** *Let  $A$  and  $B$  be  $\Sigma$ -algebras computable under  $\alpha$  and  $\beta$ , respectively. Let the  $\Sigma$ -homomorphism  $\phi : A \rightarrow B$  be  $(\alpha, \beta)$ -computable. Then the image  $im(\phi)$  of  $A$  under  $\phi$  is a  $\beta$ -computable subalgebra of  $B$ .*

**Proof.** Let  $\phi$  be tracked by the recursive function  $f$ . Then  $f(\Omega_\alpha)$  is a recursively enumerable subset of  $\Omega_\beta$ . Clearly,  $\beta(f(\Omega_\alpha))$  generates  $im(\phi)$ , i.e.  $\langle \beta(f(\Omega_\alpha)) \rangle = im(\phi)$ . By Lemma 2.4.4,  $im(\phi)$  is a  $\beta$ -computable subalgebra of  $B$ . ■

Let  $\equiv$  be a congruence on  $A$  and consider the factor algebra  $A/\equiv$ . Let  $A$  be an algebra computable under  $\alpha$ . Then composing  $\alpha$  with the natural homomorphism  $\nu : A \rightarrow A/\equiv$ , defined by  $\nu(a) = [a]$  where  $[a]$  is the equivalence class of  $a \in A$ , we obtain an epimorphism

$$\alpha^* = \nu \circ \alpha : \Omega_\alpha \rightarrow A/\equiv$$

that is an effective numbering of  $A/\equiv$ . The natural homomorphism is computable with respect to  $\alpha$  and  $\alpha^*$  with tracking function the identity.

**Lemma 2.4.9.** *Let  $A$  be a computable algebra and  $\equiv$  a congruence on  $A$ . If  $\equiv$  is  $\alpha$ -decidable,  $\alpha$ -semidecidable, or  $\alpha$ -cosemidecidable then the factor algebra  $A/\equiv$  is  $\alpha^*$ -computable,  $\alpha^*$ -semicomputable, or  $\alpha^*$ -cosemicomputable, respectively.*

**Theorem 2.4.10 (Computable Homomorphism Theorem).** *Let  $A$  and  $B$  be  $\Sigma$ -algebras computable under  $\alpha$  and  $\beta$ , respectively. Let the  $\Sigma$ -homomorphism  $\phi : A \rightarrow B$  be  $(\alpha, \beta)$ -computable. Then the image  $im(\phi)$  of  $B$  under  $\phi$  is a  $\beta$ -computable  $\Sigma$ -subalgebra of  $B$  that is computably isomorphic with  $A/\equiv_\phi$ , with respect to  $\beta$  and  $\alpha^*$  under the natural isomorphism.*

In analysing computable aspects of a bit of algebra, the situation commonly arises that the algebras of interest are semicomputable or cosemicomputable and one is investigating special algebraic properties that make some of the algebras computable. Two important examples are the properties that an algebra is



1. simple, and

2. approximated by finite algebras, or residually finite,

both of which provide arguments for cosemicomputability when working with semicomputable algebras. The relevant algebra is discussed in Meinke and Tucker [1992, Section 4].

In order to discuss simpleness we will look at the computability of congruences in more detail. Let  $R \subseteq A^2$ . The congruence  $\equiv_R$  generated by  $R$  is the smallest congruence on  $A$  containing  $R$ .

**Lemma 2.4.11.** *Let  $A$  be an algebra semicomputable under  $\alpha$  and let  $R$  be an  $\alpha$ -semidecidable subset of  $A^2$ . Then the congruence  $\equiv_R$  on  $A$  generated by  $R$  is an  $\alpha$ -semidecidable congruence on  $A$ . Furthermore,  $\equiv_R$  is semidecidable uniformly in  $R$ .*

On any algebra  $A$  there are two *trivial* congruences, namely: the *unit congruence*  $A^2$  in which all elements are identified, and the *null congruence*  $=$  in which no distinct elements are identified. An algebra  $A$  is said to be *simple* if every congruence on  $A$  is trivial.

The following result originates from Kuznetsov [1958].

**Theorem 2.4.12.** *Let  $A$  be a finitely generated algebra that is semicomputable under  $\alpha$ . If  $A$  is simple then  $A$  is computable under  $\alpha$ .*

**Proof.** Suppose that  $A$  is the unit algebra, i.e.  $|A| = 1$ . Then  $A$  is computable. Suppose that  $A$  is not the unit algebra. By hypothesis, we can enumerate the relation  $\equiv_\alpha$  since it is semidecidable. Let us next examine the complement of  $\equiv_\alpha$ .

If  $|A| \geq 2$  then  $A$  contains (at least) two distinct elements  $a_1$  and  $a_2$  with  $\alpha$ -numbers  $y_1$  and  $y_2$ , respectively.

Given any  $\alpha$ -numbers  $x_1$  and  $x_2$  we add the pair  $(x_1, x_2)$  to the congruence  $\equiv_\alpha$  and generate the smallest congruence  $\equiv_{\alpha, x_1=x_2}$  that contains  $\equiv_\alpha$  and  $(x_1, x_2)$ . By Lemma 2.4.11, this congruence is semicomputable uniformly in  $x_1$  and  $x_2$ . If  $x_1 \equiv_\alpha x_2$  then this will be discovered from the enumeration of  $\equiv_\alpha$ . However, if  $x_1 \not\equiv_\alpha x_2$  then the congruence  $\equiv_{\alpha, x_1=x_2}$  is a proper extension of the congruence  $\equiv_\alpha$  and is trivial, and hence all elements are identified, because  $A$  is simple. Thus,  $x_1 \not\equiv_\alpha x_2$  if, and only if,  $y_1 \equiv_{\alpha, x_1=x_2} y_2$ , i.e.  $(y_1, y_2)$  will appear in the enumeration of  $\equiv_{\alpha, x_1=x_2}$ . ■

The following result generalises the result on simpleness. Let  $A$  be an algebra. A congruence  $\equiv$  on  $A$  has *finite index* if  $|A/\equiv|$  is finite.

**Theorem 2.4.13.** *Let  $A$  be a finitely generated algebra that is semicomputable under  $\alpha$ . If every non-unit congruence on  $A$  has finite index then  $A$  is computable under  $\alpha$ .*

Results on the use of residual finiteness based on work in McKinsey [1943] can be found in Mal'cev [1961]; these are better studied in connection with Section 3.2 below.



## 2.5 Concluding remarks

In this chapter we wish to investigate the nature of a computable algebra and, later on, that of a computable approximation to an uncountable topological algebra. Thus, in this first section, we have focused on the simplest ideas concerning the computability of countable algebras. In particular, we have not attempted to develop the theory of computable universal algebras. Let us note some of the material we have neglected.

There is a theory of numberings of sets and algebras, which exploits much of recursion theory, including the theory of degrees.

There is a theory of the computability of general algebraic constructions, such as *direct products*, *free products*, *tensor products*, *direct limits*, *automorphism groups*, *endomorphism semigroups* etc. Ideas about the effectiveness and uniformity of algebraic constructions are essential in understanding the role of computability in algebra and have several applications, most obviously in the semantics of data type constructions. For example, the family of finitely generated subalgebras of a computable algebra  $A$  forms a computable directed system of computable algebras whose computable direct limit is the algebra  $A$ . In the theory of computable constructions the central invariance notions, such as computable stability and autostability, must be relativised. For example, we wish to express the fact that the computable numberings of a polynomial ring  $R[X]$  are unique up to recursive equivalence relative to the computable numbering of the ring  $R$ .

There are specific theories such as the theory of computable groups and the theories of computable rings, modules, and fields.

For further information about this theory of computable algebras see the articles in Mal'cev [1971], Ershov [1973; 1975; 1977b], and the book Ershov [1979].

Now we turn to the use of finitely generated computable algebras in the theory of data types.

## 3 Algebraic characterisations of computable algebras

In this section we discuss the characterisation of computable algebras by means of algebraic specification methods. The methods we use to classify computable algebras are those based on equations, and their associated term rewriting systems and initial algebra semantics. The characterisation theory was developed by J. A. Bergstra and J. V. Tucker, starting in 1979, to answer questions concerning the scope and limits of specification methods for abstract data types. In the algebraic theory of data, many sorted algebras are used to model specific data types, and equations and conditional equations are used to specify them axiomatically, uniquely up to isomorphism. Certain natural questions concerning data types arise, including:

*What data types (=algebras) can be defined by algebraic specification methods? Can algebraic specification methods define all the data types (=algebras) one wants?*

These questions lead to the formulation of *soundness*, *adequacy* and (by combination) *completeness theorems* for algebraic specification methods.

We begin with a fairly detailed explanation of the motivation in computer science for building the specification theory of computable algebras. Two of the main results of the theory are characterisations of computable many sorted algebras using

1. finite sets of equations whose term rewriting systems are complete, and
2. finite sets of equations whose initial and final algebra semantics coincide;

these are stated in full as Theorems 3.1.5 and 3.1.6, respectively. In this section we develop just enough of the single sorted theory to prove some simpler, closely related theorems.

### 3.1 Computable data types and their specification

#### Data types as algebras

Data in computations are represented by programming constructs called *data types*. In a general theory of programming we need general constructs that allow users to specify and implement their own data types. Subjects of interest are:

1. the structure of basic data types;
2. methods for building new data types from old data types;
3. the independence of a data type from its representations; and
4. the scope and limits of data type constructs and methods.

Viewed from programming, an abstract data type is a data type that can be specified and used independently of its specific implementations. For example, the integers and reals are data types that have a high degree of independence from their computer representations. The data type has been abstracted from its implementation details so that it may be specified and compiled separately from the programs that use it. The operations allowed on data are crucial in defining and using data types. Through the operations we access the data, by applying the operations to given initial data, and we understand and classify the properties of the data type.

Let us explore these ideas using algebraic models of data types.

Consider a data type  $D$  possessing a finite list of special data and operations. Let these be named in a (many sorted) finite signature  $\Sigma$ .

A specific implementation or representation of a data type  $D$  is modelled by a (many sorted) algebra  $A$  of signature  $\Sigma$ .

In many cases we assume that all the data of  $D$  can be accessed by applying the operations to the special data. This requires that each implementation  $A$  of  $D$  is generated from initial values in  $A$  named as constants in its finite signature  $\Sigma$ , i.e.  $A$  is a finitely generated minimal algebra (recall Definition 2.1.4).

The semantics of the data type  $D$  is the class of all 'acceptable' representations or implementations of the data type. This we model by *some* class  $K$  of algebras of common signature  $\Sigma$ . In many cases we take  $K$  to be the class of all minimal  $\Sigma$ -algebras.

For example, consider the data type of the integers with a maximum and minimum integer. A simple set of operations is listed in the following signature:

```
signature  integers;
sort      int;
constants  0 :→ int
           1 :→ int
           Max: → int
           Min: → int
operations +: int × int → int
           -: int × int → int
           ·: int × int → int
end
```

The data type allows many implementations. For example, there are the algebras of integers under modulo  $n$  arithmetic,

$$\mathbb{Z}_n = (\{0, 1, 2, \dots, n-1\}; 0, 1, n-1, 0, +, -, \cdot),$$

where  $n-1$  is the largest integer  $Max$  and 0 is the minimum integer  $Min$ . This is an infinite family of algebras having the properties of commutative rings with identity, and the overflow properties

$$\begin{aligned} Max + 1 &= Min, \text{ i.e. } (n-1) + 1 = 0 \text{ and} \\ Min - 1 &= Max, \text{ i.e. } 0 - 1 = (n-1). \end{aligned}$$

The class of implementations of this kind, which vary only in the size  $n$  of the maximum number, can form part of the semantics of the integers.

This structure is not as unique as it may seem. For example, in the above example for  $n > 2$ , we can take  $Max$  and  $Min$  to be any pair of numbers that satisfy the overflow conditions, such as  $Max = 0$  and  $Min = 1$  or, more generally,  $Max = k$  and  $Min = k+1$  for  $k \leq n-1$ .

Of course, many other implementations are possible, not all of which need to have the attractive commutative ring properties of  $\mathbb{Z}_n$  or satisfy the overflow properties above. There is the set  $\mathbb{Z}$  of integers with a largest

number symbol  $+\infty$  and a least number symbol  $-\infty$  adjoined, and with operations suitably modified to satisfy the overflow properties

$$\text{Max} + 1 = \text{Max} \text{ and } \text{Min} - 1 = \text{Min}.$$

There are algebras based on the set  $\{0, 1, \dots, n-1\}$  with different operations having these overflow properties.

Furthermore, the precise details of the representations of the data and operations of these algebras have not been explained. The data may be written in binary, decimal, or any other number base  $b$  and the functions defined accordingly. This observation contributes infinitely many more algebras to the class that is a possible semantics for the integers (for example, one for each base  $b$ ).

This example suggests two questions of general interest:

*What properties of the operations of the data type are used in its specification?*

*When are two implementations of the data type equivalent?*

In the case of arithmetic the classes of algebras of signature  $\Sigma$  can be classified axiomatically: for example, by means of the commutative ring axioms, the overflow properties, and other axioms. The equivalence of implementations is determined by the comparison of, say, decimal and binary data representations of the integers.

It is essential to compare implementations of data types. In the algebraic theory of data this is done by homomorphisms. In particular, two specific implementations, modelled by algebras  $A$  and  $B$ , are equivalent if they are isomorphic as algebras. This means that the properties of data types studied and conveyed to users are isomorphism invariants and are required to be common to all isomorphic implementations.

Let  $A$  be an algebra that implements a data type  $D$ . A property  $P$  of  $A$  is an *abstract property* of the data type  $D$  if for any algebra  $B$  that implements  $D$ , if  $P$  is true of  $A$  and  $B \cong A$  then  $P$  is true of  $B$ .

Considering the immense number of practically important details that are involved in implementations of data types, this is a rather liberal form of semantical abstraction. In the case of numerical data types, for example, properties of the integers that are specific to implementations in binary are not abstract properties of the integers. These ideas motivate the following:

**Definition 3.1.1.** An *abstract data type* consists of

1. a signature  $\Sigma$ ;
2. a class  $K$  of  $\Sigma$ -algebras that is closed under isomorphism, i.e.

$$\text{if } A \in K \text{ and } B \cong A \text{ then } B \in K.$$

We sometimes refer to  $A \in K$  as a *concrete data type*.



There are some useful refinements of this general definition. Often, an abstract data type is assumed to be a class closed under isomorphism in which *all* algebras are isomorphic, i.e. an *isomorphism type*. For example, the natural, integer, rational, real, and complex numbers are defined in this way by users, as are terms, trees, and strings over some given syntax. Another possible condition is that all the algebras in the class are minimal. Also, these two conditions can be combined to define an abstract data type to be the isomorphism type of a minimal algebra (see Wirsing [1990]).

At the heart of the theory is the study of algebras uniquely defined up to isomorphism.

### Computable data types

An obvious question is:

*Given that an abstract data type is modelled by a class  $K$  of algebras, which representations in  $K$  can be implemented on a computer? That is, which algebras in  $K$  are finite, computable, semicomputable, or cosemicomputable?*

An algebra  $A \in K$  represents a concrete representation of the data type whose semantics is  $K$ . Thus, we may interpret the idea that  $A$  is computable under numbering  $\alpha$  to mean that  $A$  can be encoded, translated or compiled into a concrete numerical representation  $\Omega_\alpha$  as defined by  $\alpha$ .

Let  $Comp(K)$  be the subclass of all algebras in  $K$  that are computable. Computability is an isomorphism invariant (Theorem 2.2.5) so we know that if

$$A \in Comp(K) \text{ and } B \cong A \text{ then } B \in Comp(K).$$

This means that computability is an abstract property of a data type in the sense above. Computability is a stable notion for finitely generated algebras (Theorem 2.3.14) so if  $K$  contains finitely generated algebras  $A$  and  $B$  we know that if  $A$  and  $B$  are isomorphic then  $A$  and  $B$  are computably isomorphic. This implies we can recursively translate between any pair of concrete numerical representations  $\Omega_\alpha$  and  $\Omega_\beta$  of  $A$  and  $B$ , respectively.

The important special case when  $K$  is an isomorphism type is especially well-behaved; then: *every implementation in  $K$  is computable if, and only if, one implementation in  $K$  is computable*. Further, if  $K$  contains only minimal algebras then any pair of numerical implementations can be recursively interchanged.

**Definition 3.1.2.** An abstract data type is *computable*, *semicomputable* or *cosemicomputable* if it contains an algebra  $A$  that is computable, semicomputable or cosemicomputable, respectively.

### Comparing algebras by homomorphisms

The class  $K$  of algebras which models the semantics of a data type is a rich structure for investigation. Most data types have a wide range of algebras



of interest, providing case studies and general theoretical problems. For example, in addition to numerical types such as those above, the many and varied models that can be considered implementations of the stack have been much researched (see Examples 3.2.11 below).

We reconsider the comparison of data type implementations. After reflecting on implementing algebras using algebras of natural numbers, we met the idea that algebra  $A$  implements algebra  $B$ :  $A$  is a representation of  $B$  if there is a surjective homomorphism  $\alpha : A \rightarrow B$ . (Recall Definition 1.4.1.)

The structure of the semantics of a data type can be investigated using the class  $K$  of algebras together with homomorphisms between them. In classifying the algebras/implementations of an abstract data type  $K$  in which all algebras are minimal, this idea is simpler: any homomorphism between two minimal algebras is surjective.

Generalisations of this idea are necessary to accommodate non-minimal algebras that arise in the study of the semantics of specifications. The classes of algebras and their morphisms compose a category, and many properties of the semantics of data types can be analysed at the level of category theory.

An important construct we will use is the following.

**Definition 3.1.3.** Let  $K$  be any class of algebras of signature  $\Sigma$ . An algebra  $I$  is *pre-initial* for  $K$  if for every  $A \in K$ , there is a unique  $\Sigma$ -homomorphism  $\phi : I \rightarrow A$ . If  $I \in K$  then we say that  $I$  is *initial* for class  $K$ .

Any class  $K$  has many pre-initial algebras but it need not possess any initial algebra. If  $I$  is initial for  $K$  then it is unique up to isomorphism. That is, if  $I$  and  $J$  are both initial algebras in  $K$  then  $I \cong J$ .

If  $I$  is pre-initial then  $I$  can be used as some kind of standard algebra modelling the data type, because *every* concrete implementation  $A$  of  $\text{Min}(K)$  can be represented by a unique surjective homomorphism  $\phi : I \rightarrow A$  as

$$A \cong I / \equiv_{\phi},$$

for a unique congruence  $\equiv_{\phi}$  on  $I$ .

The idea is that an initial algebra  $I$ , if it exists, models the data type semantics  $K$  in a way that is independent of methods for its implementation or specification. If the class  $K$  is an isomorphism type then  $I$  'is' the data type. The questions arise:

*What kind of data types have initial algebras?*

*When are initial algebras computable?*

### Implementation using terms

The algebra  $T(\Sigma)$  of all closed terms over non-void signature  $\Sigma$  is pre-initial for any class  $K$  of  $\Sigma$ -algebras (Lemma 2.1.9) which says that  $T(\Sigma)$  is initial

in the class  $Alg(\Sigma)$  of all  $\Sigma$ -algebras). Thus, every minimal algebra  $A$  is a unique homomorphic image of  $T(\Sigma)$ ; remember that  $\Sigma$  is finite and  $T(\Sigma)$  is computable.

The basic method for constructing an initial algebra  $I$  for  $K$ , if it exists, is to construct a factor algebra of the syntactic algebra  $T(\Sigma)$ . In fact,

$$I \cong T(\Sigma) / \equiv_K$$

where  $\equiv_K$  is a congruence for which  $t \equiv_K t'$  means that the closed terms  $t$  and  $t'$  are equivalent syntactic expressions in all the algebras of  $K$ , i.e.

$$t \equiv_K t' \text{ if, and only if, } K \models t = t'.$$

The problem of implementing and specifying the data type  $K$  via  $I$  can be investigated through the corresponding problem of implementing or specifying the congruence  $\equiv_K$ .

This algebra  $T(\Sigma) / \equiv_K$  is important for implementing the data type. Using term rewriting techniques, we may design and implement algorithms for term manipulation, especially algorithms for computing  $\equiv_K$ . A central technical idea is that of a *transversal* for the equivalence relation  $\equiv_K$ , which is a set of terms containing one and only one term for each equivalence class, and provides a set of normal forms. From the point of view of term rewriting theory, some might say that an abstract data type  $K$  is *computable* (or *semicomputable* etc.) when  $\equiv_K$  is a decidable (or semidecidable etc.) relation on  $T(\Sigma)$ .

### Algebraic specifications

To specify the class  $K$  of algebras that models the semantics of a data type, an axiomatic theory  $(\Sigma, E)$  is used so that  $K \subseteq Alg(\Sigma, E)$ . In particular, if  $E$  is a set of equations or conditional equations and  $K = Alg(\Sigma, E)$  then  $T(\Sigma) / \equiv_K \in K$ , and

$$E \vdash t = t' \Leftrightarrow t \equiv_K t'$$

where the proof system for  $\vdash$  is that for equational (or conditional equational) logic, applied to closed terms. In this case we write  $T(\Sigma, E)$  for this construction of the initial algebra  $I$  of  $Alg(\Sigma, E)$ . Thus, for any recursively enumerable  $E$  we know that  $\equiv_K$  is recursively enumerable. This means that the initial algebra  $T(\Sigma, E)$  is semicomputable. If  $\equiv_K$  is decidable then  $T(\Sigma, E)$  is computable. The specification  $(\Sigma, E)$  is desired to be finite.

The properties of the specification  $(\Sigma, E)$  that determine the decidability of the congruences for initial algebras can be analysed. The equations  $E$  may be interpreted as left-to-right rewrite rules for transforming terms of  $T(\Sigma)$ . Thus  $(\Sigma, E)$  serves both

(1) as an *axiomatic specification* for a variety of algebras; and

(2) as a *term rewriting system*, intended to formalise a system of deductions, governed by simple algebraic substitution rules, within which a deduction

$$t \rightarrow t' \text{ implies } t \equiv_K t'$$

(but not conversely).

The choice of  $(\Sigma, E)$  leads to a set  $J$  of normal forms that may be a transversal for  $\equiv_K$ : given  $t, t' \in T(\Sigma)$ , to decide whether  $t \equiv_E t'$  one uses  $E$  to calculate their prescribed 'normal forms'  $n, n' \in J$  and on completing the deductions  $t \rightarrow_E n, t' \rightarrow_E n'$  one checks  $n = n'$ .

The semantics  $K$  of an abstract data type can be uniquely determined up to isomorphism with an initial algebra  $I$ , rather than by a particular syntactical construction. The decidability of  $\equiv_K$  means that the algebra  $T(\Sigma)/\equiv_K$  is computable under our definition; in particular, since the concept of a computable algebra is an isomorphism invariant, we can remove all mention of syntax in the semantical concept of a computable abstract data type.

### Completeness theorems for equational specification methods

The question addressed here is:

*What finite algebraic specification methods exist that specify all and only the computable algebras?*

This question concerns three taxonomic notions.

**Definition 3.1.4.** Let  $M$  be a method for defining algebras uniquely up to isomorphism.

The method  $M$  is *sound* for a class  $C$  of algebras if each algebra  $A$  defined by method  $M$  is in  $C$ .

The method  $M$  is *adequate* for a class  $C$  of algebras if each algebra  $A$  in  $C$  can be defined by method  $M$ .

The method  $M$  is *complete* for a class  $C$  of algebras if  $M$  is sound and adequate for  $C$ .

It is known that the finite equational or conditional equational method will not define all computable, and hence all semicomputable, algebras; these methods are not adequate. Enriching the method to allow the use of a *finite number of hidden sorts and functions* does enable it to specify any semicomputable (and hence any computable) algebra. Thus, the algebras that can be algebraically specified using initial algebra semantics are precisely the semicomputable algebras (see Theorem 3.2.6); these methods are complete.

We examine the term rewriting properties of finite equational hidden function specifications of algebras and we give an algebraic characterisation of the computable algebras in terms of *complete* term rewriting systems. A complete term rewriting system is one whose reductions or rewrites satisfy

the Church-Rosser or confluence property and are strongly terminating or noetherian. In Section 3.5, we prove a simpler single sorted version of this theorem about many sorted algebras:

**Theorem 3.1.5 (First Completeness Theorem).** *Let  $A$  be a finitely generated minimal  $\Sigma$ -algebra. Then the following are equivalent:*

1.  $A$  is computable.
2. There is a finite equational specification  $(\Sigma_0, E_0)$  such that
  - (a)  $\text{Sort}(\Sigma) = \text{Sort}(\Sigma_0)$  and  $\Sigma \subseteq \Sigma_0$ ;
  - (b)  $(\Sigma_0, E_0)$  is a complete term rewriting system;
  - (c)  $T(\Sigma_0, E_0) \upharpoonright_{\Sigma} \cong A$ .

Furthermore, the  $(\Sigma_0, E_0)$  can be taken to be an orthogonal term rewriting system, whose size is independent of the algebra  $A$ , depending only on the signature  $\Sigma$ . More specifically, there is a fixed bound for the number of hidden functions needed in  $\Sigma_0$ , but there is no fixed bound for the number of equations needed in  $E_0$ .

The fact that (2) implies (1) is straightforward and is a principal reason for the usefulness of the complete term rewriting system. The fact that (1) implies (2) is more difficult.

The specific properties of the equational specification  $(\Sigma_0, E_0)$  constructed in the theorem suggest a number of questions. Of special interest is the fact that the size of the specification  $(\Sigma_0, E_0)$  can be large. We show in Section 3.3 that any computable algebra can be given a *very* small equational specification.

Another important method of specifying a data type is to use final algebra semantics. Here, an equational specification  $(\Sigma_0, E_0)$  can sometimes be given that has a non-trivial final object  $F(\Sigma_0, E_0)$  in  $\text{Alg}^*(\Sigma_0, E_0)$ , which is the class  $\text{Alg}(\Sigma_0, E_0)$  with the unit algebras removed.

For example, in Bergstra and Tucker [1983b], one version of this method is explained and the following theorem about many sorted algebras is proved:

**Theorem 3.1.6 (Second Completeness Theorem).** *Let  $A$  be a finitely generated minimal  $\Sigma$ -algebra. Then the following are equivalent:*

1.  $A$  is computable.
2. There is a finite equational specification  $(\Sigma_0, E_0)$  such that
  - (a)  $\text{Sort}(\Sigma) = \text{Sort}(\Sigma_0)$ ,  $|\text{Sort}(\Sigma)| = n$ , and  $\Sigma \subseteq \Sigma_0$ ;
  - (b)  $\Sigma_0$  has  $3(n+1)$  hidden functions, i.e. functions that are not in  $\Sigma$ ;
  - (c)  $E_0$  has  $2(n+1)$  equations; and
  - (d)  $T(\Sigma_0, E_0) \upharpoonright_{\Sigma} \cong A$  and  $F(\Sigma_0, E_0) \upharpoonright_{\Sigma} \cong A$ .

In particular, the size of  $(\Sigma_0, E_0)$  is independent of  $A$  and, indeed, the number of hidden functions and equations is independent of the number of



constants and operations in the signature  $\Sigma$  of  $A$ .

Can any computable algebra be given such a small, finite equational specification that is also a complete term rewriting system? There is a finite algebra such that the number of equations in an equational specification which is a complete term rewriting system may be large (Theorem 3.6.1). Thus the specification in the First Theorem above cannot be improved in this regard.

To complement the theorem further, we mention some of the term rewriting properties of computable and semicomputable algebras. For example, we show that a computable algebra can have an *infinite* recursive equational specification *without hidden functions* that is an orthogonal complete term rewriting system (Theorem 3.4.15). From this construction, we obtain a finite equational specification without hidden functions for a finite algebra that is an orthogonal complete term rewriting system (Corollary 3.4.16).

In Bergstra and Tucker [1983b], details of a specification method based on final algebra semantics were presented and the cosemicomputable algebras were shown to be precisely the algebras defined by algebraic specifications under final algebra semantics; see Meseguer *et al.* [1992] for further techniques and results.

Computable functors representing parametrisation methods for abstract data types can be specified: see Bergstra and Klop [1983] and Rodenburg [1991].

A next stage in the development of specification methods is that of higher order algebraic specification methods. This is motivated by the need to specify higher order algebras, such as algebras containing data and functions on the data, and the uncountable algebras we have seen in Section 1.3. Some basic results are given in Meinke [1992]. Naturally, the higher order methods are more powerful than first order methods. Some early results on their effectivity are in Meinke [1994] and Kosiuczenko and Meinke [1994].

## 3.2 Equational specifications

We gather together the definitions, examples, and results that are needed to understand and prove a theorem which demonstrates the adequacy of equational methods for computable algebras.

**Definition 3.2.1.** Let  $A$  be an algebra of signature  $\Sigma$ . Then  $A$  is said to have an *equational specification*  $(\Sigma, E)$ , under initial algebra semantics, if  $E$  is a set of equations over  $\Sigma$  such that

$$T(\Sigma, E) \cong A.$$

**Example 3.2.2.** Consider the algebra



$$A = (\mathbb{N}; 0, n+1, n+m, n \cdot m, n^2)$$

of natural numbers. We may specify easily this algebra by means of the (primitive recursive) equations  $(\Sigma, E)$  displayed below:

<b>signature</b>	<i>arithmetic with square;</i>
<b>sort</b>	<i>nat;</i>
<b>constant</b>	$0 : \rightarrow \text{nat}$
<b>operations</b>	<i>succ: nat <math>\rightarrow</math> nat</i> <i>add: nat <math>\times</math> nat <math>\rightarrow</math> nat</i> <i>mult: nat <math>\times</math> nat <math>\rightarrow</math> nat</i> <i>sq: nat <math>\rightarrow</math> nat</i>
<b>end</b>	
<b>equations</b>	<i>add</i> $(x, 0) = x$ <i>add</i> $(x, \text{succ}(y)) = \text{succ}(\text{add}(x, y))$ <i>mult</i> $(x, 0) = 0$ <i>mult</i> $(x, \text{succ}(y)) = \text{add}(\text{mult}(x, y), x)$ <i>sq</i> $(x) = \text{mult}(x, x)$
<b>end</b>	

To establish the correctness of the specification  $(\Sigma, E)$  for  $A$  is to show that

$$A \cong T(\Sigma, E).$$

It is a very useful exercise to prove this in detail, using the isomorphism  $\phi : A \rightarrow T(\Sigma, E)$  defined for  $n \in \mathbb{N}$  by

$$\phi(n) = [\text{succ}^n(0)].$$

**Definition 3.2.3.** An algebra  $A$  of signature  $\Sigma$  is said to have an *equational hidden enrichment specification*  $(\Sigma_0, E_0)$ , under initial algebra semantics, if  $\Sigma \subseteq \Sigma_0$ , and  $E_0$  is a set of equations over  $\Sigma_0$  such that

$$T(\Sigma_0, E_0) \upharpoonright_{\Sigma} = \langle T(\Sigma_0, E_0) \rangle_{\Sigma} \cong A.$$

In most cases the algebra  $A$  is  $\Sigma$ -minimal, in which case it is sufficient to show that

$$T(\Sigma_0, E_0) \upharpoonright_{\Sigma} \cong A.$$

**Example 3.2.4.** Consider the algebra

$$B = (\mathbb{N}; 0, n+1, n^2)$$

of natural numbers with signature  $\Sigma^{SQ}$  displayed below:

<b>signature</b>	<i>square algebra</i> ;
<b>sort</b>	<i>nat</i> ;
<b>constant</b>	$0 : \rightarrow \text{nat}$
<b>operations</b>	$\text{succ} : \text{nat} \rightarrow \text{nat}$ $\text{sq} : \text{nat} \rightarrow \text{nat}$
<b>end</b>	

The algebra  $B$  is a reduct of the algebra  $A$  in Example 3.2.2, is  $\Sigma_{SQ}$ -minimal, and

$$A \upharpoonright_{\Sigma_{SQ}} = \langle A \rangle_{\Sigma_{SQ}} = B.$$

We can specify  $B$  by specifying  $A$  using the equational specification  $(\Sigma, E)$  given earlier. Since  $A \cong T(\Sigma, E)$ , we have

$$T(\Sigma, E) \upharpoonright_{\Sigma_{SQ}} = \langle T(\Sigma, E) \rangle_{\Sigma_{SQ}} \cong B.$$

Thus,  $(\Sigma, E)$  is an equational specification of  $B$  with (two) hidden functions.

The question arises: *Can the square algebra  $B$  be given a finite equational specification without hidden functions?*

In Bergstra and Tucker [1987] it is proved (Theorem 4.1) that the answer is: no. A number of other simple structures (satisfying a property called sparsity) are shown to need hidden functions in their finite algebraic specifications. The lesson of this and many other examples is that the use of hidden functions in an algebraic specification is both natural and essential.

We assume that each specification  $(\Sigma, E)$  consists of a finite signature  $\Sigma$  and a set  $E$  of equations. We will classify specifications by means of algorithmic properties of the set  $E$ .

All standard numberings of terms are recursively equivalent. Let  $\gamma : \Omega_\gamma \rightarrow T(\Sigma, X)$  be a computable numbering for the algebra of terms over  $\Sigma$  in the indeterminates in  $X$ . From  $\gamma$  we can computably number the set  $\text{Eqn}(\Sigma, X)$  of equations in an obvious way: the numbering  $\gamma_e : \Omega_\gamma \times \Omega_\gamma \rightarrow \text{Eqn}(\Sigma, X)$  is obtained from  $\gamma$  by pairing.

**Definition 3.2.5.** We define  $E \subseteq \text{Eqn}(\Sigma, X)$  to be *recursively enumerable* if the preimage set  $\gamma_e^{-1}(E)$  of  $\gamma_e$ -codes is a recursively enumerable set of numbers.

Recall the permutation closure of a set of equations from 2.1.5. We note that if  $E$  is recursively enumerable then the permutation closure  $\hat{E}$  is recursively enumerable.

However, we define  $E \subseteq \text{Eqn}(\Sigma, X)$  to be *recursive* if  $\gamma_e^{-1}(E)$  is recursive and, in addition, if  $\gamma_e^{-1}(\hat{E})$  is recursive.

An equational specification  $(\Sigma, E)$  is *finite*, *recursive*, or *recursively enumerable* if its set  $E$  of equations is finite, recursive, or recursively enumerable.

The recursively enumerable and recursive specifications could be called semicomputable and computable, or semidecidable and decidable, following the terminology of Section 2.2. However, it is convenient in the case of syntax to use the established nomenclature.

We are interested primarily in finite specifications but the scope of the equational methods of interest are *recursively enumerable equational specifications under initial algebra semantics*. It is easy to show that these latter methods define precisely the *semicomputable algebras*.

**Lemma 3.2.6 (Basic Completeness Lemma for Semicomputable Algebras).** *Let  $A$  be a minimal  $\Sigma$ -algebra. Then the following are equivalent:*

1.  $A$  is semicomputable.
2. There is a recursively enumerable equational specification  $(\Sigma, E)$  such that

$$\models T(\Sigma, E) \cong A.$$

3. There is a recursively enumerable equational specification  $(\Sigma_0, E_0)$  such that
  - (a)  $\text{Sort}(\Sigma) \subseteq \text{Sort}(\Sigma_0)$  and  $\Sigma \subseteq \Sigma_0$ ;
  - (b)  $T(\Sigma_0, E_0) \upharpoonright_{\Sigma} \cong A$ .

**Proof.** First we show that (1) implies (2). If  $A$  is semicomputable then  $A \cong T(\Sigma) / \equiv_A$  since  $T(\Sigma)$  is pre-initial and  $\equiv_A$  is recursively enumerable by Corollary 2.3.11. We may use  $\equiv_A$  as a set of equations (without variables) in an equational specification  $(\Sigma, \equiv_A)$  of  $A$ . In this case it can be shown that

$$T(\Sigma, \equiv_A) = T(\Sigma) / \equiv_A \cong A.$$

The fact that (2) implies (3) is immediate from the definitions.

The proof that (3) implies (1) is as follows. If  $(\Sigma_0, E_0)$  is recursively enumerable then  $\equiv_{E_0}$  is recursively enumerable because for all  $t$  and  $t'$ ,

$$E_0 \vdash t = t' \text{ if, and only if, } t \equiv_{E_0} t'.$$

Thus,  $T(\Sigma_0, E_0)$  is semicomputable and hence  $T(\Sigma_0, E_0) \upharpoonright_{\Sigma}$  is semicomputable because  $A$  is  $\Sigma$ -minimal. ■

Consider a cosemicomputable finitely generated algebra that is *not* computable. It cannot have a recursively enumerable equational (or conditional equation) specification under initial algebra semantics because otherwise it would be computable. For example, in Bergstra and Tucker [1980b], such a finitely generated minimal algebra of primitive recursive functions is given a finite equational specification under final algebra semantics.

There exist semicomputable algebras that cannot be specified by *recursive* equational specifications (Bergstra and Tucker [1987, Theorem 6.5]).

We can improve on Lemma 3.2.6 by using finite specifications with hidden sorts and functions as follows.

**Theorem 3.2.7 (A Simple Completeness Theorem for Semicomputable Algebras).** *Let  $A$  be a minimal  $\Sigma$ -algebra. Then the following are equivalent:*

1.  $A$  is semicomputable.
2. There is a finite equational specification  $(\Sigma_0, E_0)$  such that
  - (a)  $\text{Sort}(\Sigma) \subseteq \text{Sort}(\Sigma_0)$  and  $\Sigma \subseteq \Sigma_0$ ;
  - (b)  $\Sigma_0$  has finitely many hidden functions, i.e. functions that are not in  $\Sigma$ ;
  - (c)  $E_0$  has finitely many equations; and
  - (d)  $T(\Sigma_0, E_0) \upharpoonright_{\Sigma} \cong A$ .

The fact that (2) implies (1) is a consequence of Basic Lemma 3.2.6, since every finite specification is recursively enumerable. The fact that (1) implies (2) is Theorem 5.3 in Bergstra and Tucker [1987] where it is sufficient to add one sort to make an equational specification of any semicomputable algebra.

This theorem can be improved in a number of ways, for example by considering the size of  $(\Sigma_0, E_0)$ , and its independence of  $A$ . We leave it as an exercise to extract further information on  $(\Sigma_0, E_0)$  from the proof in Bergstra and Tucker [1987].

The technically natural classification of semicomputable algebras is blocked by the following problem:

**Problem 3.2.8.** *Can each semicomputable algebra be given a finite equational specification under initial algebra semantics using hidden constants and functions but without using hidden sorts?*

Some partial positive results were observed in Bergstra and Tucker [1983a]; some examples of interest are contained in Bergstra and Tucker [1987]. In Kasymov [1987] the answer is shown to be: no. Semicomputable algebras are addressed in Vrancken [1987], Marongiu and Tulipani [1987; 1989], Gagliardi and Tulipani [1990], and Bergstra and Heering [1994].

There are many examples of equationally specified algebras that are semicomputable but not computable. We consider a few examples drawn from some well-understood areas.

**Examples 3.2.9 (Examples of finitely presented semigroups and groups).** Finite presentations for semigroups and groups are special types of equational specifications under initial algebra semantics. For example, the finite group presentation

$$\langle x_1, \dots, x_n \mid w_1, \dots, w_m \rangle,$$

where the  $w_1, \dots, w_m$  are group words over  $x_1, \dots, x_n$ , is equivalent to the following equational specification  $(\Sigma, E)$ :

```

signature    group presentation;
sort         grp;
constants    1 :→ grp
               $x_1$  :→ grp
              ...
               $x_n$  :→ grp
operations    $\cdot$  : grp × grp → grp
               $^{-1}$  : grp → grp

end
equations
     $(x \cdot y) \cdot z = x \cdot (y \cdot z)$ 
     $1 \cdot x = x \cdot 1$ 
     $x \cdot x^{-1} = x^{-1} \cdot x = 1$ 
     $w_1 = 1$ 
    ...
     $w_m = 1$ 

end

```

The group defined by the presentation is the algebra  $T(\Sigma, E)$ .

Finitely presented semigroups and groups are examples of semicomputable algebras that are defined by finite equational specifications using initial algebra semantics. *A finitely presented semigroup or group is computable if, and only if, it has a decidable word problem.* This was observed in Rabin [1960] and follows from the general results in Section 2.3.

Finite presentations for semigroups and groups seem to be the oldest and most extensively studied equational specifications. The word problem for semigroups was studied by A. Thue in 1914. The undecidability of the word problem for finitely presented semigroups was established in Markov [1947] and Post [1947]; in Turing [1950] the unsolvability of the word problem for semigroups with cancellation was established.

The word problem for groups was stated in Dehn [1910; 1911] and solved for the fundamental groups of certain topological spaces (closed orientable surfaces). The undecidability of the word problem for groups was first proved by P. S. Novikov in 1955 and independently by W. W. Boone in the same year. Modern constructions and proofs can be found in Rotman [1994] and in Stillwell [1982].

The theory of algorithmic properties of finite presentations and its history is involved and fascinating. There are characterisation theorems of semicomputable and computable groups that are related to the theorems and questions about semicomputable and computable universal algebras discussed earlier. For example, *Higman's Theorem* can be stated as



*a finitely generated group is semicomputable if, and only if, it is embeddable in a finitely presented group*

(Higman [1961]). Building on Kuznetsov's Theorem for algebras (Theorem 2.4.12), the *Boone-Higman Theorem* states that

*a finitely presented group is computable if, and only if, it is embeddable in a simple subgroup of a finitely presented group*

(Boone and Higman [1974]). Another characterisation is the *MacIntyre-Neumann Theorem* which states that

*a finitely generated group is computable if, and only if, it is embeddable in every countable algebraically closed group*

(MacIntyre [1972] (if) and Neumann [1973] (only if)).

The subject is surveyed conveniently in the papers by Stillwell [1982] and Miller III[1992]; and it is covered fully in the excellent monographs by Magnus *et al.*, [1976] and Lyndon and Schupp [1977]. The collections of Boone *et al.*, [1973], Adian *et al.* [1980], and Baumslag and Miller III[1992] contain further valuable information.

Lastly, consider the semigroup given in Section 1.2 (25); this is defined in our notation as follows:

**signature**    *Tzeitin's semigroup;*

**sort**             $s$ ;

**constants**     $a : \rightarrow s$

$b : \rightarrow s$

$c : \rightarrow s$

$d : \rightarrow s$

$e : \rightarrow s$

**operation**     $\cdot : s \times s \rightarrow s$

**end**

**equations**

$(x \cdot y) \cdot z = x \cdot (y \cdot z)$

$a \cdot c = c \cdot a$

$a \cdot d = d \cdot a$

$b \cdot c = c \cdot b$

$b \cdot d = d \cdot b$

$e \cdot c \cdot a = c \cdot e$

$e \cdot d \cdot b = d \cdot e$

$c \cdot c \cdot a = c \cdot c \cdot a \cdot e$

**end**

The semigroup defined is the initial algebra  $S \cong T(\Sigma, E)$  and was shown to have an undecidable word problem by G. C. Tzeitin in 1958; thus  $S$  is a semicomputable algebra that is not computable. See Lallement [1979] for an account of the word problem for semigroups, including Tzeitin's.

### Examples 3.2.10 (Recognising properties of finite presentations of semigroups and groups).

A finite equational (or conditional equation) specification of an algebra is a finite object. Given an algebraic property  $P$  of algebras, such as finiteness or computability, we can ask:

*Is there an algorithm that decides whether or not the algebra  $T(\Sigma, E)$  defined by a specification  $(\Sigma, E)$  has the property  $P$ ?*

We consider a special case of this question:

*What properties of groups are decidable given their finite presentations?*

A property of a finitely presented group is a *Markov property* if there are two finitely presented groups  $G$  and  $H$  such that

1.  $G$  has property  $P$ , and
2. if  $H$  is embedded in a finitely presented group  $K$  then  $K$  does not have property  $P$ .

The *Adian–Rabin Theorem* states that

*if  $P$  is a Markov property of finitely presented groups then  $P$  is not decidable*

(Adian [1957] and Rabin [1958]).

Thus, there is no algorithm to decide finiteness and computability, among *many* interesting properties. In the case of computability, it was shown in Boone and Rogers [1966] that computability is a property of finitely presented groups, that is  $\Sigma_3$ -complete in the Arithmetic Hierarchy.

### Examples 3.2.11 (The stack).

A stack stores data and its state may be changed by storing a new datum or retrieving a previously stored datum; it has an empty state when it is not storing data. The data and stack states are modelled by sets  $D$  and  $S$ , and the storage procedures are modelled by constant  $\emptyset$  and functions

$$push : D \times S \rightarrow S, top : S \rightarrow D \text{ and } pop : S \rightarrow S.$$

Taken together the sets, constants, and functions form a minimal many sorted algebra.

In the large literature concerned with the semantic modelling of the stack there is a consensus about

1. the initialisation of the stack using  $\emptyset$ ;
2. the construction of all possible stack states using *push*; and
3. the normal operation of *top* and *pop* when the stack is not empty.

However, there are many ways to handle

$$top(\emptyset) \text{ and } pop(\emptyset),$$

which give rise to many stack algebras and axiomatic specifications.

*What is the class of algebras that is the class of all stack algebras? Can the class be used to organise the semantic models of the stack in the literature into a coherent mathematical theory in which, for example, different stack models can be compared by means of homomorphisms? Can any stack algebra be equationally specified? Are all stack algebras computable?*

We give an equational specification  $(\Sigma_{\text{st}}, E_{\text{st}})$  whose axioms express the consensus properties of the stack. The class  $\text{Alg}_m(\Sigma_{\text{st}}, E_{\text{st}})$  of all minimal  $\Sigma_{\text{st}}$  algebras satisfying the equations in  $E_{\text{st}}$  is then taken to be the class of all stack algebras.

A stack is modelled by a minimal algebra of signature  $\Sigma_{\text{st}}$  satisfying a set  $E_{\text{st}}$  of equations, defined as follows:

<b>signature</b>	$\Sigma_{\text{st}}$
<b>sorts</b>	<i>data</i> , <i>edata</i> , <i>stack</i>
<b>constants</b>	$d_1, \dots, d_n : \rightarrow \text{data}$ $\emptyset : \rightarrow \text{stack}$
<b>operations</b>	$i : \text{data} \rightarrow \text{edata}$ $\text{push} : \text{edata} \times \text{stack} \rightarrow \text{stack}$ $\text{top} : \text{stack} \rightarrow \text{edata}$ $\text{pop} : \text{stack} \rightarrow \text{stack}$
<b>end</b>	

We assume there are at least  $n \geq 2$  constants of sort *data*, and ignore operations involving *data* only. The sort *edata* is for *extended data* which allows the data to be augmented by error elements, unspecified elements etc. The operation *i* maps data into extended data. *The stack is conceived as a stack of extended data.*

In defining the set  $E_{\text{st}}$  of equations over  $\Sigma_{\text{st}}$  that we assume true of all stacks we must avoid specifying the behaviour of the operations in 'debatable' circumstances. First, we define a sequence of standard stack polynomials.

Let  $a_1, a_2, \dots$  be a fixed list of variables of type *data*. The *standard stack polynomials* over these variables are inductively defined by

$$\begin{aligned} T_0 &= \emptyset \\ T_{k+1} &= \text{push}(i(a_{k+1}), T_k) \end{aligned}$$

for  $k \in \mathbb{N}$ .

Thus, the sequence begins

$$\emptyset, \text{push}(i(a_1), \emptyset), \text{push}(i(a_2), \text{push}(i(a_1), \emptyset)), \dots$$

These polynomials represent standard stack states containing data and no extended data. Notice that  $T_k$  represents a standard stack state containing  $k$  elements of sort *data*.

The equations in  $E_{st}$  are for  $k \in \mathbb{N}$ ,

stack equations

$$\begin{aligned} top(T_{k+1}) &= i(a_{k+1}) & t-eqn_{k+1} \\ pop(T_{k+1}) &= T_k & p-eqn_{k+1} \end{aligned}$$

end

Thus  $E_{st}$  is an infinite equational specification and the sequence begins

$$\begin{aligned} top(push(i(a_1), \emptyset)) &= i(a_1) & t-eqn_1 \\ pop(push(i(a_1), \emptyset)) &= \emptyset & p-eqn_1 \\ top(push(i(a_2), push(i(a_1), \emptyset))) &= i(a_2) & t-eqn_2 \\ pop(push(i(a_2), push(i(a_1), \emptyset))) &= push(i(a_1), \emptyset) & p-eqn_2 \\ &\dots \end{aligned}$$

The equations leave unspecified the effects of the operations on  $\emptyset$  and on any extended data. Notice that the first  $2k$  equations for any  $k = 1, 2, \dots$  in the list specify that the *top* and *pop* operations behave as expected on all standard stack states containing  $k$  elements of sort *data*.

**Definition 3.2.12.** A *stack algebra* is a minimal algebra in the equationally defined class  $Alg(\Sigma_{st}, E_{st})$ .

The infinitary equational specification  $(\Sigma_{st}, E_{st})$  is an orthogonal complete term rewriting system (see Section 3.4 below). The set  $N(\Sigma_{st}, E_{st})$  of normal forms of the term rewriting system is decidable, and there is a unique total computable function  $n$  that computes normal forms.

**Theorem 3.2.13.** *The initial algebra*

$$T(\Sigma_{st}, E_{st})$$

of the variety  $Alg(\Sigma_{st}, E_{st})$  is computable. There is a finite set  $E_R$  of equations over  $\Sigma_{st}$  such that the stack algebra  $T(\Sigma_{st}, E_{st} \cup E_R)$  is semicomputable but not computable.

The *provability problem* for equations over  $(\Sigma_{st}, E_{st})$ , i.e. the problem

given any equation  $e$  over  $\Sigma_{st}$ , is  $E_{st} \vdash e$ ?

is decidable. (This can be shown using the fact that the equational theory  $(\Sigma_{st}, E_{st})$  is  $\omega$ -complete.) However, the existence of  $E_R$  shows that it is undecidable whether or not given any closed equation  $e$  over  $\Sigma_{st}$ ,

$$E_{st} \cup E_R \vdash e.$$

This  $E_R$  can be chosen to make the deduction problem a complete r.e. set. Hence, the *deduction problem* for equations over  $(\Sigma_{st}, E_{st})$ , i.e. the problem given any equations  $e_1, \dots, e_k, e$  over  $\Sigma_{st}$ , is  $E_{st} \cup \{e_1, \dots, e_k\} \vdash e$ ?

is undecidable. A consequence is that the first-order theory of the stack is undecidable.

Interestingly, the initial model  $T(\Sigma_{\text{st}}, E_{\text{st}})$  does *not* possess a finite equational specification  $(\Sigma_{\text{st}}, E)$ . However, consider the set  $C_{\text{st}}$  of four conditional equations over  $\Sigma_{\text{st}}$ :

**stack conditional equations**

$$\begin{array}{ll} \text{top}(\text{push}(i(a), \emptyset)) = i(a) & t - eqn_1 \\ \text{pop}(\text{push}(i(a), \emptyset)) = \emptyset & p - eqn_1 \\ \text{top}(x) = i(a) \rightarrow \text{top}(\text{push}(i(b), x)) = i(b) & t - ceqn \\ \text{top}(x) = i(a) \rightarrow \text{pop}(\text{push}(i(b), x)) = x & p - ceqn \end{array}$$

**end**

Then  $(\Sigma_{\text{st}}, C_{\text{st}})$  is a finite conditional equation specification for  $T(\Sigma_{\text{st}}, E_{\text{st}})$  and

$$T(\Sigma_{\text{st}}, C_{\text{st}}) = T(\Sigma_{\text{st}}, E_{\text{st}}).$$

There is also a useful finite equational specification  $(\Sigma_{\text{ast}}, E_{\text{ast}})$  with a hidden sort and two hidden operations that specifies the initial algebra  $T(\Sigma_{\text{st}}, E_{\text{st}})$ ; furthermore, the specification is an orthogonal complete term rewriting system.

The above results are contained in Bergstra and Tucker [1993a]; a related survey of the stack literature is Bergstra and Tucker [1990].

### 3.3 Adequacy theorem

Consider the following theorem that shows that equational methods are sufficiently powerful to define concisely all the computable data types.

**Theorem 3.3.1 (Adequacy Theorem).** *Let  $A$  be an infinite many sorted minimal algebra of signature  $\Sigma$ . Suppose that  $|\text{Sort}(\Sigma)| = n$ , and the number of sorts with finite carriers in  $A$  is  $n_{\text{finite}} (\leq n)$ ; also let  $\Sigma$  contain  $p$  constants and  $q$  operations. If  $A$  is computable then there is a finite equational specification  $(\Sigma_0, E_0)$  such that*

1.  $\text{Sort}(\Sigma) = \text{Sort}(\Sigma_0)$  and  $\Sigma \subseteq \Sigma_0$ ;
2.  $\Sigma_0$  has  $n$  hidden constants and  $3(n+1)$  hidden functions, i.e. functions that are not in  $\Sigma$ ;
3.  $E_0$  has  $17 + p + q + 4(n-1) + n_{\text{finite}}$  equations; and
4.  $T(\Sigma_0, E_0) \upharpoonright_{\Sigma} \cong A$ .

*Note that the size of  $(\Sigma_0, E_0)$  is independent of  $A$  and dependent only on the signature  $\Sigma$  of  $A$ .*

This theorem is an early result that suggested investigating the size and complexity of equational specifications for infinite computable algebras: see Bergstra and Tucker [1993b]. The case of finite algebras is of independent interest and is the subject of Bergstra and Tucker [1993c]. We obtain specifications for finite algebras in Corollary 3.4.16 below. The techniques



involved in the cases of finite and infinite algebras were further developed and combined to prove the Second Completeness Theorem 3.1.6.

In this section we will prove the above theorem in the single sorted case.

**Theorem 3.3.2 (Adequacy Theorem for the Single Sorted Case).**

*Let  $A$  be any single sorted infinite computable minimal algebra. Suppose that  $A$  has  $p$  constants and  $q$  operations. Then  $A$  possesses a finite equational specification with one hidden constant, six hidden operations, and  $17 + p + q$  equations.*

**Proof.** Let  $A$  be any infinite computable minimal algebra with signature  $\Sigma$ . By the Representation Lemma 2.3.9, we can take  $A$  to be isomorphic with a recursive number algebra

$$R = (\mathbb{N}; c_1, \dots, c_p, f_1, \dots, f_q)$$

whose carrier is the set  $\mathbb{N}$  of all natural numbers. We concentrate on providing  $R$  with a finite equational hidden enrichment specification with the claimed size. We do this by adding one constant and six recursive functions to  $R$  to construct a new recursive number algebra  $R_0$  of signature  $\Sigma_0$  such that

$$R_0|_{\Sigma} = \langle R_0 \rangle_{\Sigma} = R.$$

Then we show that  $R_0$  has a finite equational specification  $(\Sigma_0, E_0)$  in which  $E_0$  has  $17 + p + q$  equations. The choice of the hidden functions and hence the structure of  $E_0$  is determined by the following construction.

**Enumeration of graphs of operations**

Let  $f : \mathbb{N}^k \rightarrow \mathbb{N}$  be a total recursive function. The graph of  $f$ ,

$$\text{graph}(f) = \{(x_1, \dots, x_k, f(x_1, \dots, x_k)) \in \mathbb{N}^{k+1} : x_1, \dots, x_k \in \mathbb{N}\},$$

is recursively enumerable. By the *Diophantine Theorem*, each recursively enumerable set is the projection over the natural numbers of an integer polynomial: there exists an integer polynomial

$$P_f \in \mathbb{Z}[X_1, \dots, X_k, X_{k+1}, Y_1, \dots, Y_d]$$

such that

$$\begin{aligned} \text{graph}(f) = \{(x_1, \dots, x_k, x_{k+1}) \in \mathbb{N}^{k+1} : \exists y_1, \dots, y_d \in \mathbb{N}. \\ P_f(x_1, \dots, x_{k+1}, y_1, \dots, y_d) = 1\}. \end{aligned}$$

See Matiyasevich [1993] for a full account of this result.

We wish to remove all reference to the set  $\mathbb{Z}$  of integers in this enumeration of  $\text{graph}(f)$ . Let  $X = (X_1, \dots, X_k, X_{k+1})$  and  $Y = (Y_1, \dots, Y_d)$ .

By combining those monomials  $a_\lambda m_\lambda(X, Y)$  whose coefficients  $a_\lambda \in \mathbb{Z}$  are positive and negative, respectively, we separate  $P_f$  into two natural number polynomials

$${}^+p_f, {}^-p_f \in \mathbb{N}[X_1, \dots, X_k, X_{k+1}, Y_1, \dots, Y_d],$$

such that

$$\begin{aligned} \text{graph}(f) = \{ & (x_1, \dots, x_k, x_{k+1}) \in \mathbb{N}^{k+1} : \exists y_1, \dots, y_d \in \mathbb{N}. \\ & [{}^+p_f(x_1, \dots, x_{k+1}, y_1, \dots, y_d) - {}^-p_f(x_1, \dots, x_{k+1}, y_1, \dots, y_d) \\ & = 1]\}. \end{aligned}$$

Now, we define the function  $h_f : \mathbb{N}^{k+1+d} \rightarrow \{0, 1\}$  by

$$h_f(x, y) = \begin{cases} 0 & \text{if } {}^+p_f(x, y) - {}^-p_f(x, y) \neq 1; \\ 1 & \text{if } {}^+p_f(x, y) - {}^-p_f(x, y) = 1. \end{cases}$$

In particular: for any  $y = (y_1, \dots, y_d)$ ,

$$\text{if } h_f(x_1, \dots, x_k, x_{k+1}, y) = 1 \text{ then } f(x_1, \dots, x_k) = x_{k+1}.$$

Therefore, for all  $x = (x_1, \dots, x_k, x_{k+1}), y = (y_1, \dots, y_d)$

$$h_f(x, y) \cdot f(x_1, \dots, x_k) = x_{k+1} \cdot h_f(x, y). \quad (*)$$

Notice that the function  $h_f$  is definable as a polynomial function

$$h_f(x, y) = \min({}^+p_f(x, y) - {}^-p_f(x, y), 2) \bmod 2$$

over the following list  $\Lambda$  of functions, and a constant, over  $\mathbb{N}$ :

$$0, x + 1, x + y, x - y, x \cdot y, \min(x, 2), x \bmod 2. \quad (\Lambda)$$

### Construction of $R_0$

We set  $R_0$  to be the algebra  $R$  expanded by adding all the functions in the list  $\Lambda$ . Let the signature of  $R_0$  be  $\Sigma_0$ . Let the list of names contained in  $\Sigma_0$  for the functions of the list  $\Lambda$  be the following:

$$0, \text{SUCC}, \text{ADD}, \text{DIFF}, \text{MULT}, \text{MIN}_2, \text{MOD}_2.$$

This forms a subsignature  $\Gamma$  of  $\Sigma_0$ .

### Equations

Here is a finite set  $E_0$  of equations over  $\Sigma_0$  to specify  $R_0$ . First, for each constant  $\underline{c} \in \Sigma$  naming numerical constant  $c$  in  $R$ , take the equation

*Constants*       $\underline{c} = SUCC^c$  (0)

Next we add the following equations to define the functions in the list  $\Lambda$ :

*Addition*       $ADD(X, 0) = X$  (1)  
 $ADD(X, SUCC(Y)) = SUCC(ADD(X, Y))$

*Subtraction*       $DIFF(X, 0) = X$  (2)  
 $DIFF(0, Y) = 0$   
 $DIFF(X, SUCC(Y)) = DIFF(DIFF(X, Y), SUCC(0))$

*Multiplication*       $MULT(X, 0) = 0$  (3)  
 $MULT(X, SUCC(Y)) = ADD(MULT(X, Y), X)$

*Minimum*       $MIN_2(0) = 0$  (4)  
 $MIN_2(SUCC(0)) = SUCC(0)$   
 $MIN_2(SUCC^2(Y)) = SUCC^2(0)$

*Modulus*       $MOD_2(0) = 0$  (5)  
 $MOD_2(SUCC(0)) = SUCC(0)$   
 $MOD_2(SUCC^2(Y)) = MOD_2(Y)$

*Zero*       $MULT(X, 0) = MULT(0, X) = 0$  (6)

*Unity*       $MULT(X, 1) = MULT(1, X) = X$  (7)

Finally, we add an equational translation of  $(*)$  for each  $F \in \Sigma$  naming operation  $f$  of  $R$ . For each recursive operation  $f$  of  $R$  we make some  $h_f$ . To say  $h_f$  is a polynomial function over the list  $\Lambda$  of functions of course means  $h_f$  is the map  $\mathbb{N}^{k+1+d} \rightarrow \mathbb{N}$  defined by some formal polynomial or  $\Sigma_0$ -term  $H_f \in T_{\Sigma_0}[X_1, \dots, X_k, X_{k+1}, Y_1, \dots, Y_d]$ ; we abbreviate this by

$$H_f(X, Y) \in T_{\Sigma_0}[X, Y].$$

For each  $h_f$  we choose a term  $H_f$  and add the equation

*Operation of R*  
 $MULT(H_f(X, Y), F(X_1, \dots, X_k)) = MULT(X_{k+1}, H_f(X, Y)).$  (8)

This is all of  $E_0$ . It contains  $17 + p + q$  equations.

### Correctness of the specification

We must now prove that  $T(\Sigma_0, E_0) \cong R_0$ . Let  $\equiv$  abbreviate  $\equiv_{E_0}$  and write elements of  $T(\Sigma_0, E_0) = T(\Sigma_0)/\equiv$  in the usual form  $[t]$  for  $t \in T(\Sigma_0)$ . We claim that the map  $\phi: R_0 \rightarrow T(\Sigma_0, E_0)$  defined for  $n \in \mathbb{N}$  by

$$\phi(n) = [SUCC^n(0)]$$

is an isomorphism. That  $\phi$  is bijective follows from this lemma:

**Lemma 3.3.3.** *The set  $\{SUCC^n(0) : n \in \mathbb{N}\}$  is a transversal for  $\equiv$ .*

**Proof.** We leave to the reader the task of checking that

$$SUCC^n(0) \equiv SUCC^m(0) \Leftrightarrow n = m,$$

using the fact that  $R_0$  is a model for  $E_0$ . To show that for  $t \in T(\Sigma_0)$  there is some  $SUCC^n(0)$  so that  $t \equiv SUCC^n(0)$  we use induction on the complexity of  $t$ . ■

In the basis case the term  $t$  is a constant of  $\Sigma_0$  and the lemma is immediate from the equations in  $E_0$  of type (0).

In order to prove that  $\phi$  is a homomorphism later on, we do the induction step in the form of this lemma.

**Lemma 3.3.4.** *Let  $\underline{\lambda}$  be any  $k$ -ary operation symbol of  $\Sigma_0$  naming the function  $\lambda$  of  $R_0$ . Let  $s_1, \dots, s_k \in T(\Sigma_0)$ . If  $s_i \equiv SUCC^{z_i}(0)$  for  $1 \leq i \leq k$  then*

$$\underline{\lambda}(s_1, \dots, s_k) \equiv SUCC^{\lambda(z_1, \dots, z_k)}(0).$$

**Proof.** Let  $t = \underline{\lambda}(SUCC^{z_1}(0), \dots, SUCC^{z_k}(0))$ . By considering cases for  $\underline{\lambda} \in \Sigma_0$  we show that  $t \equiv SUCC^{\lambda(z_1, \dots, z_k)}(0)$ . It is routine to do this, but we must consider first the operation symbols of  $\Gamma$ , in order, and then the rest of the operation symbols of  $\Sigma$ . The first non-trivial case is addition,  $\underline{\lambda} = ADD$ .

Here  $t = ADD(SUCC^u(0), SUCC^v(0))$ , say, and we argue by induction on  $v$ . The basis  $v = 0$  is immediate from equation (1). So assume the lemma is true for  $ADD$  for  $v = k$  and consider  $v = k + 1$ . We calculate:

$$\begin{aligned} ADD(SUCC^u(0), SUCC^v(0)) &\equiv ADD(SUCC^u(0), SUCC(SUCC^k(0))) \\ &\quad \text{since } v = k + 1; \\ &\equiv SUCC(ADD(SUCC^u(0), SUCC^k(0))) \\ &\quad \text{by equation (1);} \\ &\equiv SUCC(SUCC^{u+k}(0)) \\ &\quad \text{by induction on } k; \\ &\equiv SUCC^{u+k+1}(0) \\ &\equiv SUCC^{u+v}(0). \end{aligned}$$

Now for each  $\underline{\lambda} \in \Gamma$  the cases follow the same pattern though the case of  $ADD$ , just proven, is used as a lemma for multiplication; we omit these details and consider case  $\underline{\lambda} = F \in \Sigma$ .

Substituting  $SUCC^{z_i}(0)$  for  $1 \leq i \leq k$  and an arbitrary list  $r = (r_1, \dots, r_d) \in T(\Sigma_0)$  into equation (8) results in this identity, wherein  $z = (z_1, \dots, z_k) \in \mathbb{N}^k$ :

$$\begin{aligned} &MULT(H_f(SUCC^{z_1}(0), \dots, SUCC^{z_k}(0), SUCC^{f(z)}(0), r), \\ &F(SUCC^{z_1}(0), \dots, SUCC^{z_k}(0))) \equiv MULT(SUCC^{f(z)}(0), \\ &H_f(SUCC^{z_1}(0), \dots, SUCC^{z_k}(0), SUCC^{f(z)}(0), r)). \end{aligned}$$

Thanks to the multiplication equation (3) and equations (6) and (7), it is sufficient to prove there exists  $r$  such that

$$H_f(SUCC^{z_1}(0), \dots, SUCC^{z_k}(0), SUCC^{f(z)}(0), r) \equiv SUCC(0).$$

Since there exist  $y_1, \dots, y_d \in \mathbb{N}$  such that  $h_f(z, f(z), y_1, \dots, y_d) = 1$  we choose  $r$  to be the  $n$ -tuple of terms  $SUCC^{y_1}(0), \dots, SUCC^{y_d}(0)$  whence the identity follows from a new lemma:

**Lemma 3.3.5.** *Let  $\tau \in T_\Gamma[X_1, \dots, X_n]$  define the function  $\psi : \mathbb{N}^n \rightarrow \mathbb{N}$ . Then for all  $SUCC^{z_1}(0), \dots, SUCC^{z_n}(0) \in T(\Gamma)$ , substituting into  $\tau(X_1, \dots, X_n)$  we obtain*

$$\tau(SUCC^{z_1}(0), \dots, SUCC^{z_n}(0)) \equiv SUCC^{\psi(z_1, \dots, z_n)}(0).$$

**Proof.** We argue by induction on the complexity of  $\tau$ . The basis is trivial as  $\tau$  is either 0 or  $X_i$ .

Assume as induction hypothesis that the lemma is true of all polynomials over  $\Gamma$  of complexity lower than  $\tau(X)$ . Let  $\tau(X) = \underline{\lambda}(\tau_1(X), \dots, \tau_k(X))$  where  $\underline{\lambda} \in \Gamma$  names function  $\lambda \in \Lambda$  and  $X = (X_1, \dots, X_n)$ . Let  $\tau_i$  define  $\psi_i : \mathbb{N}^n \rightarrow \mathbb{N}$ , so the induction hypothesis says that

$$\tau_i(SUCC^{z_1}(0), \dots, SUCC^{z_n}(0)) \equiv SUCC^{\psi_i(z_1, \dots, z_n)}(0)$$

for  $1 \leq i \leq k$ . To complete the proof we must consider all  $\underline{\lambda} \in \Gamma$ .

For example, let  $\underline{\lambda} = ADD$ . Then  $\tau(X) = ADD(\tau_1(X), \tau_2(X))$  and  $\psi(z_1, \dots, z_n) = \psi_1(z_1, \dots, z_n) + \psi_2(z_1, \dots, z_n)$ . Clearly,

$$\begin{aligned} &\tau(SUCC^{z_1}(0), \dots, SUCC^{z_1}(0)) \\ &\quad \equiv ADD(SUCC^{\psi_1(z_1, \dots, z_n)}(0), SUCC^{\psi_2(z_1, \dots, z_n)}(0)) \\ &\quad \equiv SUCC^{\psi_1(z_1, \dots, z_n) + \psi_2(z_1, \dots, z_n)}(0) \end{aligned}$$

by the already proven case of addition of Lemma 3.3.4. The other cases proceed exactly in the same way.

This completes the proofs of Lemma 3.3.4 and 3.3.5.

To check  $\phi$  is a homomorphism we use Lemma 3.3.4: let  $\lambda$  be any operation of  $R_0$ ,

$$\begin{aligned} \phi(\lambda(x_1, \dots, x_k)) &= [SUCC^{\lambda(x_1, \dots, x_k)}(0)] && \text{by definition of } \phi; \\ &= [\underline{\lambda}(SUCC^{x_1}(0), \dots, SUCC^{x_k}(0))] && \text{by Lemma 3.3.4;} \end{aligned}$$



$$\begin{aligned}
&= \lambda([SUCC^{x_1}(0)], \dots, [SUCC^{x_k}(0)]) \\
&\quad \text{by definition of } \lambda \text{ on } T(\Sigma_0, E_0); \\
&= \lambda(\phi(x_1), \dots, \phi(x_k)) \quad \text{by definition of } \phi.
\end{aligned}$$

This completes the proof of the corollary.  $\blacksquare$

### 3.4 Equational specifications and term rewriting

Reasoning with equations is based on some simple rules that have a rich and intricate theory, and many applications. We will describe briefly the concepts concerning term rewriting that we need for the completeness theorem that we will prove in Section 3.5. For a detailed treatment of most of the material of this section we recommend Klop [1992]; the Completeness Theorem was also discussed there (where it is Theorem 2.4.8) but it was not proved. We begin by defining a single sorted *algebraic* reduction or replacement system which adds algebraic structure to the idea of a reduction or replacement system.

#### Definition 3.4.1 (Reduction systems).

Let  $A$  be a non-empty set. A *reduction system* or *replacement system* on  $A$  is a reflexive and transitive binary relation

$$\rightarrow_R$$

on  $A$ . For  $a, b \in A$ , if  $a \rightarrow_R b$  we say  $a$  *reduces* to  $b$  (under  $\rightarrow_R$ ), or that  $b$  is a *reduct* of  $a$  (under  $\rightarrow_R$ ).

An element  $a \in A$  is a *normal form* for  $\rightarrow_R$  if there is no  $b$  such that  $a \rightarrow_R b$ . The set of all normal forms for  $\rightarrow_R$  is denoted  $NF(\rightarrow_R)$ .

The reduction system  $\rightarrow_R$  is *Church-Rosser* or *confluent* if for any  $a \in A$  if there are  $b_1, b_2 \in A$  so that  $a \rightarrow_R b_1$  and  $a \rightarrow_R b_2$  then there is  $c \in A$  so that  $b_1 \rightarrow_R c$  and  $b_2 \rightarrow_R c$ .

The reduction system  $\rightarrow_R$  is *weakly terminating* or *weakly normalising* if for each  $a \in A$  there is some normal form  $b \in A$  so that  $a \rightarrow_R b$ .

The reduction system  $\rightarrow_R$  is *strongly terminating*, *strongly normalising*, or *noetherian* if there does not exist an infinite chain

$$a_0 \rightarrow_R a_1 \rightarrow_R \dots \rightarrow_R a_n \rightarrow_R \dots$$

of reductions in  $A$  wherein for  $i \in \mathbb{N}$ ,  $a_i \neq a_{i+1}$ .

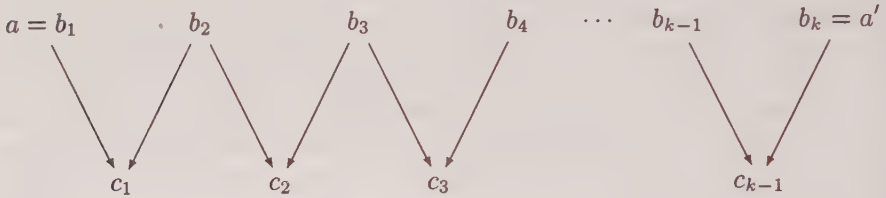
The reduction system  $\rightarrow_R$  is *complete* if it is Church-Rosser and strongly terminating.

A *reduction system* is *Church-Rosser* and *weakly terminating* if, and only if, every element reduces to a unique normal form. Clearly strong termination entails weak termination.

Let  $\equiv_R$  denote the smallest equivalence relation on  $A$  containing  $\rightarrow_R$ . It is an easy exercise to show that for  $a, a' \in A$ ,

$a \equiv_R a' \Leftrightarrow$  there is a sequence  $a = b_1, \dots, b_k = a'$  such that for each pair  $b_i, b_{i+1} \in A$  there exists a common reduct  $c_i \in A$  for  $1 \leq i \leq k-1$ .

This is depicted below



Using this characterisation of  $\equiv_R$  it is straightforward to prove this fact:

**Lemma 3.4.2.** *The reduction system  $\rightarrow_R$  on  $A$  is Church-Rosser if, and only if, for any  $a, a' \in A$  if  $a \equiv_R a'$  then there is  $c \in A$  so that  $a \rightarrow_R c$  and  $a' \rightarrow_R c$ .*

Let  $A$  be a non-empty set. Let  $\equiv$  be an equivalence relation on  $A$ . Recall that a *transversal* for  $\equiv$  is a set  $J \subseteq A$  such that

1. for each  $a \in A$  there is some  $t \in J$  such that  $t \equiv a$ ; and
2. if  $t, t' \in J$  and  $t \equiv t'$  then  $t = t'$ .

**Lemma 3.4.3.** *Let  $\rightarrow_R$  be a Church-Rosser weakly terminating reduction system on  $A$ . Then the set of normal forms  $NF(\rightarrow_R)$  is a transversal for  $\equiv_R$ .*

**Proof.** Since every element  $a \in A$  reduces to some normal form  $n \in NF(\rightarrow_R)$ , the set  $NF(\rightarrow_R)$  contains representatives for each equivalence class of  $\equiv_R$ . To check uniqueness, let  $n, m \in NF(\rightarrow_R)$  and assume  $n \equiv_R m$ . By Lemma 3.4.2, there is  $c \in A$  so that  $n \rightarrow_R c$  and  $m \rightarrow_R c$ , but since  $n, m$  are normal forms  $n = c, m = c$  and so  $n = m$ . ■

**Definition 3.4.4.** Suppose now that  $A$  is a  $\Sigma$ -algebra. Then by an *algebraic reduction system*  $\rightarrow_R$  on the algebra  $A$  we mean a reduction system  $\rightarrow_R$  on the carrier of  $A$  which is closed under the operations of  $A$  in the following sense: for each  $k$ -ary operation  $\sigma_A : A^k \rightarrow A$  of  $A$ , and for all  $a_i, b_i \in A$  and  $1 \leq i \leq k$ ,

$$a_1 \rightarrow_R b_1, \dots, a_k \rightarrow_R b_k \Rightarrow \sigma_A(a_1, \dots, a_k) \rightarrow_R \sigma_A(b_1, \dots, b_k).$$

**Lemma 3.4.5.** *If  $\rightarrow_R$  is an algebraic reduction system on an algebra  $A$  then  $\equiv_R$  is a congruence on  $A$ .*

Next we explain how a reduction system is generated by a set of *one-step reductions* and how these sets of one-step reductions can be determined from quite arbitrary sets.

Let  $X \subseteq A \times A$  and write  $(a, b) \in X$  as  $a \rightarrow_X b$ . Let  $\rightarrow_R$  be a reduction system on the set  $A$ . Then  $X$  is said to *generate*  $\rightarrow_R$  as a set of one-step reductions if (a)  $X$  is reflexive and (b)  $\rightarrow_R$  is the smallest transitive set containing  $X$ , i.e. the *transitive closure* of  $X$ .

Let  $\rightarrow_R$  be an algebraic reduction system on an algebra  $A$ . Then  $X \subseteq A \times A$  is said to *generate*  $\rightarrow_R$  as a set of algebraic one-step reductions if (a)  $X$  is reflexive; (b)  $X$  is closed under *unit substitutions* in the following sense: for each  $k$ -ary operation  $\sigma_A : A^k \rightarrow A$  of  $A$ , for any  $a, b, a_1, \dots, a_k \in A$ , and all  $1 \leq i \leq k$ , if  $a \rightarrow_X b$  then

$$\sigma_A(a_1, \dots, a_{i-1}, a, a_{i+1}, \dots, a_k) \rightarrow_R \sigma_A(a_1, \dots, a_{i-1}, b, a_{i+1}, \dots, a_k);$$

and (c)  $\rightarrow_R$  is the transitive closure of  $X$ .

In the case that  $A$  is a set, the transitive closure of any reflexive set is a reduction system. In the case that  $A$  is an algebra, the transitive closure of any reflexive set that is closed under unit substitutions can be shown to be an algebraic reduction system. Thus, in either case, starting with an *arbitrary* set  $D \subseteq A \times A$  one can construct (using closures under reflexivity and unit substitutions) a one-step reduction relation  $\rightarrow_{D(1)}$  containing it; and then its transitive closure will be a set-theoretic or algebraic reduction system  $\rightarrow_D$ , respectively.

### Definition 3.4.6 (Term rewriting systems).

Let  $T(\Sigma, X)$  be the algebra of terms over  $\Sigma$  in the set  $X$  of variables. Let  $E \subseteq Eqn(\Sigma, X)$  be a set of equations such that for each  $t = t' \in E$  the LHS  $t$  is not a variable. A substitution instance of the LHS of an equation is called a *redex* (short for reducible expression).

We define a set  $D(E) \subseteq T(\Sigma) \times T(\Sigma)$  by

$$D(E) = \{(t(r_1, \dots, r_k), t'(r_1, \dots, r_k)) : t = t' \in E \text{ and } r_i \in T(\Sigma)\}$$

and so obtain the smallest set  $\rightarrow_{E(1)}$  of algebraic one-step reductions containing  $D(E)$ , and the algebraic reduction relation  $\rightarrow_E$  it generates. This formalises the use of equations in derivations of terms in  $T(\Sigma)$  where the reduction  $t \rightarrow_E t'$  requires substitutions to be made in some equation  $e \in E$  and the LHS of  $e$  is replaced by the RHS of  $e$  in  $t$  to obtain  $t'$ .

The pair  $(\Sigma, E)$  is an *equational term rewriting system*, or *equational TRS*, for short.

The first set of properties of a term rewriting system  $(\Sigma, E)$  is now defined by applying the properties of reduction systems to  $\rightarrow_E$ . For example, the term rewriting system  $(\Sigma, E)$  is *complete* if the reduction system  $\rightarrow_E$  on  $T(\Sigma)$  is Church–Rosser and strongly terminating.

We denote by  $NF(\Sigma, E)$  the set of all normal forms of  $\rightarrow_E$  and by  $\equiv_E$  the congruence associated to  $\rightarrow_E$ . Let  $T(\Sigma, E) = T(\Sigma) / \equiv_E$ .

**Lemma 3.4.7.**  *$T(\Sigma, E)$  is the initial algebra of  $Alg(\Sigma, E)$ .*

**Lemma 3.4.8.** *Let  $A$  be a minimal  $\Sigma$ -algebra. Let  $(\Sigma, E)$  be a weakly terminating TRS. If  $A \models E$  and  $NF(\Sigma, E)$  is a transversal for  $\equiv_A$  then*

$$T(\Sigma, E) = T(\Sigma) / \equiv_A \cong A.$$

**Proof.** We shall show that for  $t, t' \in T(\Sigma)$ ,

$$t \equiv_E t' \Leftrightarrow t \equiv_A t'.$$

Case (i) Suppose  $t \equiv_E t'$ . Then  $E \vdash t = t'$  and since  $A \in Alg(\Sigma, E)$  we have  $E \models t = t'$ , by soundness. This means that  $v_A(t) = v_A(t')$  in  $A$ , i.e.  $t \equiv_A t'$ .

Case (ii) Suppose  $t \equiv_A t'$ . Since the TRS  $(\Sigma, E)$  is weakly terminating we can calculate its normal forms for all terms using a function  $nf_E : T(\Sigma) \rightarrow T(\Sigma)$ . We know that

$$t \equiv_E nf_E(t) \text{ and } t' \equiv_E nf_E(t'). \quad (*)$$

By the result of case (i), using  $A \models E$ ,

$$t \equiv_A nf_E(t) \text{ and } t' \equiv_A nf_E(t').$$

Since  $t \equiv_A t'$ ,

$$nf_E(t) \equiv_A nf_E(t'),$$

but since  $NF(\Sigma, E)$  is a transversal for  $\equiv_A$  this implies

$$nf_E(t) = nf_E(t').$$

Thus,

$$nf_E(t) \equiv_E nf_E(t')$$

and by (\*)

$$t \equiv_E t'.$$

■

**Definition 3.4.9.** The term rewriting system  $(\Sigma, E)$  is *left linear* if for all  $t = t' \in E$ , each variable that appears in  $t$  does so only once.

The term rewriting system  $(\Sigma, E)$  is *non-overlapping* if

- (i) for any pair of different equations  $t = t', r = r' \in E$ , the terms  $t$  and  $r$  do not overlap in the following sense: there exist closed substitutions

- $\tau, \rho$  of 5,  $r$  such that  $\rho(r)$  is a subterm of  $\tau(t)$  and the outermost function symbol of  $\rho(r)$  occurs as a part of  $t$ .
- (ii) for any rule  $t = t' \in E$ ,  $t$  does not overlap with itself in the following sense: there exist closed substitutions  $\tau, \rho$  of  $t$  such that  $\tau(t)$  is a proper subterm of  $\rho(t)$  and the outermost function symbol of  $\tau(t)$  occurs as a part of  $t$ .

The term rewriting system  $(\Sigma, E)$  is *orthogonal* if it is left linear and non-overlapping.

The stack specification in Example 3.2.11 is an example of an orthogonal term rewriting system. For basic information about orthogonal term rewriting systems see Section 3 of Klop [1992], where the following fact is Theorem 3.1.2:

**Lemma 3.4.10.** *If  $(\Sigma, E)$  is an orthogonal TRS then it is Church–Rosser.*

Recall the algebraic specification methods described in Section 3.2. The structural properties of a specification  $(\Sigma, E)$ , such as the Church–Rosser and termination properties, are obtained from those of its reduction relation  $\rightarrow_E$  on  $T(\Sigma)$ . For example, recalling Definition 3.2.3:

**Definition 3.4.11.** An algebra  $A$  of signature  $\Sigma$  is said to have a *finite equational complete TRS hidden enrichment specification*  $(\Sigma_0, E_0)$  if  $\Sigma \subseteq \Sigma_0$  and  $E_0$  is a finite set of equations over  $T(\Sigma, X)$  such that  $\rightarrow_E$  is a complete TRS and

$$T(\Sigma_0, E_0) \upharpoonright_{\Sigma} = \langle T(\Sigma_0, E_0) \rangle_{\Sigma} \cong A.$$

Although hidden functions are natural and essential in the application of equational term rewriting systems, not much seems to be known about the role of hidden functions in term rewriting. There exist  $\Sigma$ -algebras such that for all complete TRS equational specifications  $(\Sigma_0, E_0)$  with hidden functions the normal forms for terms over  $\Sigma$  must involve those hidden functions.

A starting point for the investigation of effectivity are these facts:

**Lemma 3.4.12 (Basic Lemma).** *Let  $(\Sigma, E)$  be a finite equational reduction system specification. Then*

1. *the set  $D_E$  and the one-step reduction relation  $\rightarrow_{E(1)}$  are decidable;*
2. *the reduction system  $\rightarrow_E$  and the congruence  $\equiv_E$  are semidecidable sets;*
3. *the set  $NF(\Sigma, E)$  of normal forms is decidable.*

*In particular,  $T(\Sigma, E)$  is a semicomputable algebra. If  $(\Sigma, E)$  is recursive or recursively enumerable then (1) and (2) hold, but the set  $NF(\Sigma, E)$  is cosemidecidable.*

Notice that  $NF(\Sigma, E)$  need not be a transversal for  $\equiv_E$ . We may now



prove the following key fact:

**Theorem 3.4.13.** *Let  $(\Sigma, E)$  be a finite equational term rewriting system specification which is Church–Rosser and weakly terminating. Then  $T(\Sigma, E)$  is a computable algebra.*

**Proof.** Given  $t \in T(\Sigma)$  we can interleave the algorithms enumerating  $NF(\Sigma, E)$  and  $\equiv_E$  to seek the normal form of  $t$  which is guaranteed to exist from the weak termination hypothesis. Given  $t, t' \in T(\Sigma)$ , to decide  $t \equiv_E t'$  we calculate their normal forms  $n, n'$  and, using the uniqueness property of Church–Rosser systems, we have only to check whether or not  $n = n'$ . ■

We will now prove some results about specifications of algebras without hidden functions.

**Corollary 3.4.14.** *Let  $A$  be a minimal algebra of signature  $\Sigma$  having a finite equational complete TRS hidden enrichment specification  $(\Sigma_0, E_0)$ . Then  $A$  is computable.*

**Proof.** By hypothesis,  $\Sigma \subseteq \Sigma_0$  and  $E_0$  is a finite set of equations over  $T(\Sigma, X)$  such that  $\rightarrow_E$  is a complete TRS and

$$T(\Sigma_0, E_0) \upharpoonright_{\Sigma} = \langle T(\Sigma_0, E_0) \rangle_{\Sigma} \cong A.$$

By Theorem 3.4.13, we know that  $T(\Sigma_0, E_0)$  is computable. Thus  $A$  is computable by invariance and the properties of the reducts. ■

Finally, we will prove a useful result that helps set the scene for the main theorem in the next section.

**Theorem 3.4.15.** *Let  $A$  be a minimal  $\Sigma$  algebra. If  $A$  is computable then  $A$  possesses a (possibly infinite) recursive equational specification  $(\Sigma, E)$  that is an orthogonal complete term rewriting system.*

**Proof.** Now  $A \cong T(\Sigma)/\equiv_A$  and since  $A$  is computable we have that  $\equiv_A$  is recursive in the standard numbering  $\gamma$  of terms. Let  $nf_{\gamma} : T(\Sigma) \rightarrow T(\Sigma)$  compute the smallest normal forms for  $\equiv_A$  as determined by the enumeration  $\gamma$ ; it is defined by

$$nf_{\gamma}(t) = \gamma((\text{least } i)[t \equiv_A \gamma(i)]).$$

Clearly  $nf_{\gamma}$  is  $\gamma$ -computable. Define

$$E_A = \{t = nf_{\gamma}(t) : t \not\equiv nf_{\gamma}(t) \text{ and for each proper subterm } r \text{ of } t, \\ nf_{\gamma}(r) = r\}.$$

Then  $T_{\gamma} = im(nf_{\gamma})$  is a transversal for  $\equiv_A$ . We claim that  $T_{\gamma}$  is the set of normal forms for the TRS  $(\Sigma, E_A)$ . First, note that the terms in  $T_{\gamma}$  do

not reduce since neither they nor their subterms occur on an LHS. Next, note that the terms not in  $T_\gamma$  can be reduced. To see this, let  $t \notin T_\gamma$  and let  $r$  be a minimal subterm of  $t$  such that  $r \notin T_\gamma$  (perhaps  $r$  is  $t$ ). Then  $r$  occurs on an LHS of an equation in  $E_A$ , and we may reduce  $t$ . Thus,  $NF(\Sigma, E_A) = T_\gamma$ .

Next we show that  $(\Sigma, E_A)$  is strongly terminating. Because the Gödel numbering  $\gamma$  of  $T(\Sigma)$  is monotonic, any reduction of  $t \rightarrow t'$  entails  $\gamma^{-1}(t) > \gamma^{-1}(t')$ . Thus, an infinitely long reduction cannot occur because  $(\mathbb{N}, <)$  is well-founded.

We show that  $(\Sigma, E_A)$  is an orthogonal TRS. Clearly, since each rule has no variables, the TRS is left linear. If two rules overlap then, since the rules have closed terms on the LHS, one of the LHS must be a subterm of the other. By inspection of  $E_A$  we see this is not the case. Thus the TRS is non-overlapping. Thus  $(\Sigma, E_A)$  is complete since it is strongly terminating and orthogonal.

Now  $(\Sigma, E_A)$  is recursive since  $E_A$  is recursive using the computability of syntactic identity  $\equiv$ ,  $n f_\gamma$ , and subterm decomposition, and  $E_A$  is trivially closed under the renaming of its variables.

Finally, since  $NF(\Sigma, E_A) = T_\gamma$  is a transversal for  $\equiv_A$  and trivially  $A \in Alg(\Sigma, E_A)$ , we deduce that

$$A \cong T(\Sigma, E_A)$$

by Lemma 3.4.8. ■

**Corollary 3.4.16.** *Let  $A$  be a  $\Sigma$ -minimal algebra. If  $A$  is finite then  $A$  possesses a finite orthogonal complete TRS specification  $(\Sigma, E)$ .*

**Proof.** It is sufficient to check that  $E_A$  is finite when  $A$  is finite. ■

Consider the size of  $(\Sigma, E)$  in the case of Corollary 3.4.16. Let

$$\begin{aligned} l &= |A| \\ n &= \text{maximum arity of the operations in } \Sigma = \max\{k : \Sigma_k \neq \emptyset\} \\ m &= |\Sigma| = 1 + |\cup\{\Sigma_k : k \geq 0\}|. \end{aligned}$$

Then  $|E| < ml^n$ .

### 3.5 Proof of First Completeness Theorem

We will prove the following single sorted version of the First Completeness Theorem 3.1.5; we leave the bound on the number of hidden functions as an exercise.

**Theorem 3.5.1 (First Completeness Theorem for the Single Sorted Case).** *Let  $A$  be a minimal  $\Sigma$ -algebra. Then the following are equivalent:*

1.  $A$  is computable.

2. There is a finite equational specification  $(\Sigma_0, E_0)$  such that
- (a)  $\Sigma$  and  $\Sigma_0$  have the same sort and  $\Sigma \subseteq \Sigma_0$ ;
  - (b)  $(\Sigma_0, E_0)$  is a complete term rewriting system;
  - (c)  $T(\Sigma_0, E_0) \upharpoonright_{\Sigma} \cong A$ .

Furthermore, the  $(\Sigma_0, E_0)$  may be taken to be an orthogonal term rewriting system, whose size is independent of the algebra  $A$ , depending only on the signature.

**Proof.** The fact that statement (2) implies statement (1) was established in Corollary 3.4.14; we prove that (1) implies (2).

Let  $\Sigma$  be a single sorted signature with  $p$  constants and  $q$  operations. Let  $A$  be any computable minimal  $\Sigma$ -algebra. If  $A$  is finite then there exists an appropriate finite equational specification  $(\Sigma, E)$  for  $A$  by Corollary 3.4.16.

Suppose  $A$  is infinite. Since  $A$  is computable, by the Representation Lemma 2.3.9,  $A$  is isomorphic to a recursive algebra

$$R = (\mathbb{N}; c_1, \dots, c_p, f_1, \dots, f_q)$$

of numbers and we can concentrate on building an appropriate finite equational specification for  $R$ , which will also be a specification for  $A$ .

First, we build an expansion  $R_0$  of  $R$  by adding constants and functions such that

$$R_0 \upharpoonright_{\Sigma} = \langle R_0 \rangle_{\Sigma} = R.$$

Next, we build a finite equational specification  $(\Sigma_0, E_0)$  for  $R_0$  which we will prove is a complete TRS. This specification for  $R_0$  does not involve hidden sorts or functions.

The specification  $(\Sigma_0, E_0)$  serves as an appropriate specification for the  $\Sigma$ -algebra  $R$  and involves hidden functions.

The choice of hidden functions and hence the structure of  $E_0$  is determined by the following construction which is designed to compute each operation  $f$  of  $R$  in a special way.

### Enumeration functions

Let  $\phi : \mathbb{N}^k \rightarrow \mathbb{N}$  be any total recursive function. Then, by the *Kleene Normal Form Theorem*, this may be written

$$\phi(x) = U((\text{least } z)T_k(e, x, z))$$

where  $U$  and  $T$  are the *Kleene computation function* and  $T$  *predicate*, respectively,  $e$  is some index for  $\phi$ ,  $x \in \mathbb{N}^k$ , and  $z \in \mathbb{N}$ . Since  $U$  and  $T_k$  are primitive recursive so are the functions

$$\begin{aligned} h(z, x) &= U((\text{least } z' \leq z)[z' = z \text{ or } T_k(e, x, z')]) \\ g(z, x) &= \begin{cases} 0 & \text{if } (\exists z' \leq z)T_k(e, x, z') \\ 1 & \text{otherwise.} \end{cases} \end{aligned}$$

From these functions we can define a recursive function  $t$  that simulates the least number search operation by

$$\begin{aligned} t(z, x, 0) &= h(z, x) \\ t(z, x, y + 1) &= t(z + 1, x, g(z + 1, x)). \end{aligned}$$

Thus,  $\phi$  is factorised into  $t, h, g$ ; for example, in the sense that

$$\phi(x) = t(0, x, 1)$$

(among many choices of values of  $z$  and  $y$ ). Here,  $\phi$  is computed by finding the least  $z'$  such that  $T_k(e, x, z')$ . The reader can consult Cutland [1980] and Phillips [1992] for details of the  $T$ -predicate.

### Construction of $R_0$

To make  $R_0$  we add to  $R$  the following constants and functions. We assume that each of the  $p$  constants has the form

$$c \in \mathbb{N}$$

and that each of the  $q$  operations are total recursive functions on  $\mathbb{N}$  having the form

$$f : \mathbb{N}^k \rightarrow \mathbb{N}.$$

We apply the above method for  $\phi$  to each function  $f$  on  $R$  and add the constructed functions  $h, g$ , and  $t$ .

We add the constant zero  $0 \in R$  and the successor function  $x + 1$ .

Finally, for each operation  $f$  of  $R$ , we add each primitive recursive subfunction  $\lambda$  defined in some primitive recursive definition of its primitive recursive enumeration functions  $g, h$ . Let  $\Delta_i$  be this list of all the subfunctions of the enumeration functions  $g_i$  and  $h_i$  of  $f_i$ . Such a list is added for each of the  $q$  operations of  $R$ .

### Size

Now  $R_0$  is the algebra obtained by adding all the above functions to  $R$ . Clearly,

$$R_0|_{\Sigma} = \langle R_0 \rangle_{\Sigma} = R.$$

We count these additions. Since  $\Sigma$  has  $p$  constants and  $q$  operations, to make the signature  $\Sigma_0$  we must add the following:

- 1 constant,
- 1 successor,
- $3q$  enumeration functions of the form  $h, g, t$  for operation  $f$ , and
- $\Pi_{fun}$  subfunctions of the  $h$  and  $g$ ,

where  $\Pi_{fun} = \Pi_1 + \dots + \Pi_q$  and  $\Pi_i$  is the number of subfunctions in the list  $\Delta_i$  of primitive recursive subfunctions of  $g_i$  and  $h_i$  associated with the

$i$ -th operation  $f_i$ . Note that  $\Pi_{fun}$  is dependent on the algebra  $R$ , whereas the other numbers are independent of the algebra and depend only on the number of constants and functions in  $\Sigma$ . (Later we will show how the dependency of  $\Pi_{fun}$  on  $R$  can be removed.)

### Construction of $(\Sigma_0, E_0)$

We now define the specification  $(\Sigma_0, E_0)$  for  $R_0$ . Let signature  $\Sigma$  have the form

```

sort            $s$ 
constants     ...
                 $\underline{c} : \rightarrow s$ 
                ...
operations    ...
                 $F : s^k \rightarrow s$ 
                ...
end

```

where there are  $p$  constants and  $q$  operations.

Then  $\Sigma_0$  is  $\Sigma$  with the following names for the new operations adjoined:

```

constants      $0 : \rightarrow s$ 
operations     $SUCC : s \rightarrow s$ 
                ...
                 $G : s \times s^k \rightarrow s$       for each operation  $F$  of  $\Sigma$ 
                 $H : s \times s^k \rightarrow s$ 
                 $T : s \times s^k \times s \rightarrow s$ 
                 $\underline{\Delta}$ 
                ...
end

```

Here  $\underline{\Delta}$  is a list of names for the functions in the list  $\Delta$  of subfunctions in the definition of  $G$  and  $H$ . We will often use  $\underline{\lambda}$  for a notation naming a function  $\lambda$  in  $R_0$ .

The equations of  $E_0$  are constructed as follows. For each constant symbol  $\underline{c}$  naming  $c \in R$ .

$$\text{Constants} \quad \underline{c} = SUCC^c(0). \quad (0)$$

For each operation  $f$  of  $R$  we add equations for the enumeration functions  $g, h$ , and  $t$ , and the primitive recursive subfunctions  $\Delta$  of  $g$  and  $h$ , as follows. Recalling the definition of function  $t$ , we add

*Enumeration*

$$T(Z, X, 0) = H(Z, X) \quad (1)$$

$$T(Z, X, SUCC(Y)) = T(SUCC(Z), X, G(SUCC(Z), X)) \quad (2)$$

$$F(X) = T(0, X, SUCC(0)). \quad (3)$$

For each primitive recursive function  $\lambda \in \Delta \cup \{g, h\}$  we add equations for its name  $\underline{\lambda}$  by means of the following method:



*Zero* If  $\lambda(x_1, \dots, x_k) = 0$  is the defining equation for  $\lambda$  then add

$$\underline{\lambda}(X_1, \dots, X_k) = 0. \quad (4)$$

*Projection* If  $\lambda(x_1, \dots, x_k) = x_i$  is the defining equation for  $\lambda$  then add

$$\underline{\lambda}(X_1, \dots, X_k) = X_i. \quad (5)$$

*Successor* If  $\lambda(y) = y + 1$  is the defining equation for  $\lambda$  then add

$$\underline{\lambda}(Y) = SUCC(Y). \quad (6)$$

*Composition* If  $\lambda(x) = \mu(\mu_1(x), \dots, \mu_n(x))$  is the defining equation for  $\lambda$ , where  $x = (x_1, \dots, x_k)$ , then add

$$\underline{\lambda}(X) = \underline{\mu}(\underline{\mu}_1(X), \dots, \underline{\mu}_n(X)) \quad (7)$$

where  $X = (X_1, \dots, X_k)$ .

*Primitive recursion* If  $\lambda(0, x) = \mu_1(x)$  and  $\lambda(y + 1, x) = \mu_2(y, x, \lambda(y, x))$  are the defining equations for  $\lambda$  then add

$$\underline{\lambda}(0, X) = \underline{\mu}_1(X) \quad (8)$$

$$\underline{\lambda}(SUCC(Y), X) = \underline{\mu}_2(Y, X, \underline{\lambda}(Y, X)) \quad (9)$$

where, again,  $x$  and  $X$  are possibly  $k$ -tuples.

Let us count the equations in  $E_0$ : there are

$p$  for the constants

$3q$  for the functions of type  $t$  and operations

$\Pi_{eqn}$  for the functions of type  $g, h$ , and  $\Delta$ .

The method for primitive recursive functions adds at most two equations per function; using the previous notation, we note that

$$\Pi_{eqn} < 2.2q + 2.\Pi_{fun} = 2(2q + \Pi_{fun}).$$

Thus, the number of equations depends on the structure of (the enumeration of) the operations.

Notice that  $R_0$  satisfies  $E_0$ .

**Lemma 3.5.2 (Correctness of specification).** *The finite equational specification  $(\Sigma_0, E_0)$  is an initial algebra specification of  $R_0$ , i.e.*

$$R_0 \cong T(\Sigma_0, E_0).$$

As a TRS,  $(\Sigma_0, E_0)$  has the following properties:

1. it is orthogonal;

2. it is strongly terminating;
3. the set of normal forms is

$$NF(\Sigma_0, E_0) = \{SUCC^n(0) : n = 0, 1, \dots\}.$$

From (1) and (2), it follows that  $(\Sigma_0, E_0)$  is a complete TRS.

**Proof.** Note that  $R_0$  satisfies  $E_0$  by construction. First we will show the statement (1) and then (3). Next, we will assume (2) and deduce that  $R_0 \cong T(\Sigma_0, E_0)$ . Finally, we prove (2).

1. The TRS  $(\Sigma_0, E_0)$  is left linear and non-overlapping by inspection.
3. Define

$$T = \{SUCC^n(0) : n = 0, 1, \dots\}.$$

Clearly,  $T \subseteq NF(\Sigma_0, E_0)$  because no equation of  $E_0$  has an LHS that matches with the numerals of  $T$ .

Now we consider the converse inclusion. Let  $t \notin T$ ; we show that  $t$  reduces and so is not a normal form.

Let  $r$  be a minimal subterm of  $t$  such that  $r \notin T$ . There are two cases.

- (a) If  $r = c$  then there is a reduction by equation (0) for constants.
- (b) If  $r = \underline{\lambda}(t_1, \dots, t_k)$  for  $\underline{\lambda}$  any function symbol then  $t_1, \dots, t_k$  must be numerals by the minimality of  $r$ . For each choice of  $\underline{\lambda}$  there is a reduction of  $r$  by means of an equation of  $E_0$  and so  $t$  is not a normal form.

We prove the isomorphism  $R_0 \cong T(\Sigma_0, E_0)$  under the assumption that (2) holds, i.e. that the TRS  $(\Sigma_0, E_0)$  is strongly terminating. We define a map  $\phi : R_0 \rightarrow T(\Sigma_0, E_0)$  by

$$\phi(x) = [SUCC^x(0)]$$

for all  $x \in \mathbb{N}$ . Since the numerals are normal forms by 3.5.2 (3), the map is injective. If the strong termination property 3.5.2 (2) holds then every term has a normal form that is a numeral, and so the map is surjective.

We show  $\phi$  is a homomorphism.

Let  $\underline{\lambda}$  be any operation symbol in  $\Sigma_0$  of type  $s^k \rightarrow s$  naming the operation  $\lambda$  in  $R_0$ ; we show that

$$\phi(\lambda(x_1, \dots, x_k)) = \underline{\lambda}(\phi(x_1), \dots, \phi(x_k)).$$

The LHS is

$$[SUCC^{\lambda(x_1, \dots, x_k)}(0)].$$

The RHS is

$$\underline{\lambda}([SUCC^{x_1}(0)], \dots, [SUCC^{x_k}(0)]) = [\underline{\lambda}(SUCC^{x_1}(0), \dots, SUCC^{x_k}(0))],$$

by definition of  $\underline{\lambda}$  in  $T(\Sigma_0, E_0)$ .

To show that these terms are equivalent in  $E_0$  we reason as follows. Note that since  $R_0$  is an algebra of numbers,

$$R_0 \models \text{SUCC}^{\lambda(x_1, \dots, x_k)}(0) = \underline{\lambda}(\text{SUCC}^{x_1}(0), \dots, \text{SUCC}^{x_k}(0)).$$

If the strong termination property 3.5.2 (2) holds, since the numerals are the normal forms, we have

$$E_0 \vdash \underline{\lambda}(\text{SUCC}^{x_1}(0), \dots, \text{SUCC}^{x_k}(0)) = \text{SUCC}^y(0)$$

for some  $y \in \mathbb{N}$ . Thus, because  $R_0$  satisfies  $E_0$ ,

$$R_0 \models \text{SUCC}^y(0) = \text{SUCC}^{\lambda(x_1, \dots, x_k)}(0)$$

and, by uniqueness,  $y = \lambda(x_1, \dots, x_k)$ . Therefore,

$$E_0 \vdash \underline{\lambda}(\text{SUCC}^{x_1}(0), \dots, \text{SUCC}^{x_k}(0)) = \text{SUCC}^{\lambda(x_1, \dots, x_k)}(0),$$

and the equivalence classes are identified, and  $\phi$  is a homomorphism.

### Proof of strong termination

Finally, we prove (2) of Lemma 3.5.2, that each  $t \in T(\Sigma_0)$  is strongly terminating, by induction on the complexity of  $t$ .

As basis we consider all constant symbols. If  $t = 0$  then we know that no reduction is possible from  $E_0$  because it is a normal form.

If  $t = \underline{c}$  names the constant  $c \in R$  then by inspection of  $E_0$  there is at most one reduction possible, by means of equation (0), and this leads to a normal form  $\text{SUCC}^c(0)$ .

The induction step we formulate as follows.

**Lemma 3.5.3.** *Let  $s_1, \dots, s_k \in T(\Sigma_0)$  be strongly terminating terms and let  $\underline{\lambda}$  be a  $k$ -ary function symbol of  $\Sigma_0$ . Then  $\underline{\lambda}(s_1, \dots, s_k)$  is a strongly terminating term.*

**Proof.** We prove this by induction on an ordering of the function symbols.

First, we order the signature  $\Sigma_0$  by ordering the operations of  $R_0$ . For each operation  $f_i$  of  $R$  let  $h_i, g_i, t_i$  be the functions factoring it and let  $\Delta_i$  be the list of primitive recursive functions used in the definitions of the  $h_i$  and  $g_i$ , those of  $h_i$  preceding those of  $g_i$ , and each of these two lists ordered by the complexity of the primitive recursive definitions of the  $h_i$  and  $g_i$  respectively. Thus, we order the constants and operations of  $R_0$  into the list

$$0, \text{succ}, \Delta_1, \dots, \Delta_q, h_1, \dots, h_q, g_1, \dots, g_q, t_1, \dots, t_q, f_1, \dots, f_q$$

and we let the signature  $\Sigma_0$  of  $R_0$  be ordered in this way. We shall now prove the lemma by induction on the position of  $\underline{\lambda}$  in this ordering of  $\Sigma_0$ .

The proof divides into a basis case, and an induction step involving nine cases, many of which split into subcases. In every case we argue by contradiction. We assume that an infinite reduction

$$t \rightarrow t_1 \rightarrow t_2 \rightarrow \dots \rightarrow t_N \rightarrow t_{N+1} \rightarrow \dots$$

exists for  $t = \underline{\lambda}(s_1, \dots, s_k)$  and  $s_1, \dots, s_k$  strongly terminating. We assume that  $t_N \rightarrow t_{N+1}$  is the first reduction where a rule is used that involves the outermost function symbol  $\underline{\lambda}$ . Such an  $N \in \mathbb{N}$  must exist for otherwise the infinite reduction would contradict the assumption that the  $s_1, \dots, s_k$  are strongly terminating. Thus  $t_1, \dots, t_N$  have the forms

$$t_i \equiv \underline{\lambda}(r_{i1}, \dots, r_{ik})$$

for  $i = 1, \dots, N$ , and for some  $1 \leq \alpha(i) \leq k$ ,

$$\begin{aligned} r_{ij} &= r_{i+1j} & \text{for } j \neq \alpha(i) \\ r_{ij} &\rightarrow r_{i+1j} & \text{for } j = \alpha(i). \end{aligned}$$

Note that  $t_N = \underline{\lambda}(r_{N1}, \dots, r_{Nk})$  and that the subterms  $r_{N1}, \dots, r_{Nk}$  are strongly terminating because the  $s_1, \dots, s_k$  are.

We will consider such reduction sequences for every type of operator  $\underline{\lambda}$  in the ordered list. For convenience, we would like to assume that  $N = 1$  and that the first reduction involves  $\underline{\lambda}$ ; this is possible:

**Lemma 3.5.4.** *The following are equivalent:*

1. *For all strongly terminating terms  $r_1, \dots, r_k$  each reduction sequence of  $\underline{\lambda}(r_1, \dots, r_k)$  is finite and  $\underline{\lambda}(r_1, \dots, r_k)$  is strongly terminating.*
2. *For all strongly terminating terms  $r_1, \dots, r_k$  each reduction sequence of  $\underline{\lambda}(r_1, \dots, r_k)$  that begins with a reduction involving  $\underline{\lambda}$  is finite.*

**Proof.** That (1) implies (2) is immediate. Consider the converse. Consider the arbitrary infinite reduction sequence

$$t \rightarrow t_1 \rightarrow t_2 \rightarrow \dots \rightarrow t_N \rightarrow t_{N+1} \rightarrow \dots$$

If this is infinite then the reduction sequence

$$t_N \rightarrow t_{N+1} \rightarrow \dots$$

is infinite and involves  $\underline{\lambda}$  at the first step. By (2) we deduce that this is impossible. Hence  $t$  is strongly terminating. ■

Thus, in each argument, we will examine the reduction

$$t \rightarrow t_1$$

and show that  $t_1$  is strongly terminating (often it cannot be further reduced).

We now begin the proof by induction on the position of  $\underline{\lambda}$  in the list.

**Basis  $\underline{\lambda}$  is *SUCC* and  $t = \text{SUCC}(s)$ .**

There is no equation in  $E_0$  that allows us to rewrite *SUCC*, so an infinite reduction sequence from  $t$  starting with such a rewrite is impossible.

**Induction steps** We assume that the statement of Lemma 3.5.3 is true for all function symbols preceding  $\underline{\lambda}$  in the list. There are three cases and several subcases.

**Case 1  $\underline{\lambda}$  names a primitive recursive function on  $R$  from**

$$\Delta_1, \dots, \Delta_q, h_1, \dots, h_q, g_1, \dots, g_q.$$

Note that  $\underline{\lambda}$  is constructed from functions earlier in the list. There are five subcases corresponding to the five parts of the definition of primitive recursion. We will classify these cases using the function named by  $\underline{\lambda}$ .

**Subcase 1a: Constant function**  $\underline{\lambda}$  names function  $\lambda$  with defining equation  $\lambda(x_1, \dots, x_k) = 0$ .

If  $t = \underline{\lambda}(s_1, \dots, s_k)$  then the only equation that applies is (4) and so  $t_1 = 0$  and is a normal form. The sequence halts and  $t$  is strongly terminating.

**Subcase 1b: Projection**  $\underline{\lambda}$  names function  $\lambda$  with defining equation  $\lambda(x_1, \dots, x_k) = x_j$ .

If  $t = \underline{\lambda}(s_1, \dots, s_k)$  then the only equation that applies is (5). Hence  $t_1 = s_j$  which is strongly terminating by hypothesis.

**Subcase 1c: Successor**  $\underline{\lambda}$  names function  $\lambda$  with defining equation  $\lambda(y) = y + 1$ .

If  $t = \underline{\lambda}(s)$  then the only equation that applies is (6). Hence  $t_1 = \text{SUCC}(s)$  which is strongly terminating by the earlier basis case of this lemma. Hence  $t$  is strongly terminating.

**Subcase 1d: Composition**  $\underline{\lambda}$  names function  $\lambda$  with defining equation

$$\lambda(x) = \mu(\mu_1(x), \dots, \mu_n(x))$$

where  $x = (x_1, \dots, x_k)$ .

If  $t = \underline{\lambda}(s)$  where  $s = (s_1, \dots, s_k)$  then the only equation that applies is (7). Then

$$t_1 = \underline{\mu}(\underline{\mu}_1(s), \dots, \underline{\mu}_n(s))$$

where  $\underline{\mu}, \underline{\mu}_1, \dots, \underline{\mu}_n$  name the subfunctions of  $\lambda$ . Since the  $\underline{\mu}, \underline{\mu}_1, \dots, \underline{\mu}_n$  precede  $\underline{\lambda}$  in the list, by the main induction hypothesis we know that  $\underline{\mu}_1(s), \dots, \underline{\mu}_n(s)$  are strongly terminating and, further, that  $t_1$  is strongly terminating; hence so is  $t$ .

**Subcase 1e: Primitive recursion**  $\underline{\lambda}$  names the function  $\lambda$  with defining equations



$$\begin{aligned}\lambda(0, x) &= \mu_1(x) \\ \lambda(y + 1, x) &= \mu_2(y, x, \lambda(y, x))\end{aligned}$$

where  $x = (x_1, \dots, x_k)$ .

Let  $t = \underline{\lambda}(r, s)$  where  $s = (s_1, \dots, s_k)$ . The only equations that apply are (8) and (9). We show that  $t$  is strongly terminating by induction on the value  $v(r)$  in  $R_0$ . (Recall that  $v$  is defined in Section 2.1.)

If  $v(r) = 0$  then the only equation that applies is (8) and  $t_1 = \underline{\mu}_1(s)$ . Since  $\underline{\mu}_1$  occurs before  $\underline{\lambda}$  we know that  $\underline{\mu}_1(s)$  is strongly terminating and so is  $t$ .

If  $v(r) > 0$  then we take as induction hypothesis the following: for all strongly terminating  $r, s$  such that  $v(r) = d$  the term  $\underline{\lambda}(r, s)$  is strongly terminating.

Let  $t = \underline{\lambda}(r, s)$  for  $v(r) = d + 1$ . The only equation that applies is (9). Thus,  $r = \text{SUCC}(z)$  for some  $z$  and

$$t_1 = \underline{\mu}_2(z, s, \underline{\lambda}(z, s)).$$

By the induction hypothesis on  $v(r)$ , we know that  $\underline{\lambda}(z, s)$  is strongly terminating, because  $v(z) = d$ . Therefore, by the main induction hypothesis of 3.5.3, since  $\underline{\mu}_2$  precedes  $\underline{\lambda}$  in the list, and  $z, s, \underline{\lambda}(z, s)$  are strongly terminating, we know that  $t_1$  and hence  $t$  are strongly terminating.

**Case 2**  $\lambda$  is  $T_j$  and  $t = T_j(r, s, u)$  where  $s = (s_1, \dots, s_k)$ .

The only equations that can be applied to the first reduction are (1) and (2). Define

$$\chi(r, s) = (\text{least } z)[g_j(z, v(s)) = 0] - v(r).$$

We argue using induction on  $\chi(r, s)$ .

**Basis**  $\chi(r, s) = 0$ .

**Subcase a:**  $v(u) = 0$ . Then  $u = 0$  and the equation used is (1), and  $t_1 = H_j(r, s)$ . Since  $H_j$  precedes  $T_j$  in the list, by induction, since  $r$  and  $s$  are strongly terminating so  $t_1$  and (hence)  $t$  are strongly terminating.

**Subcase b:**  $v(u) \neq 0$ . Then the equation used is (2) and

$$t_1 = T_j(\text{SUCC}(r), s, G_j(\text{SUCC}(r), s)).$$

First note that

$$\text{SUCC}(r), s, \text{ and } G_j(\text{SUCC}(r), s)$$

are strongly terminating by hypothesis or by induction on the list. It follows that an infinite reduction of  $t_1$  must involve a further application of equations (1) and (2). For this to happen  $G_j(\text{SUCC}(r), s)$  must reduce to either 0 or to  $\text{SUCC}(z)$  for some  $z$ .

**Claim**  $v(G_j(SUCC(r), s)) = 0$ .

**Proof of claim.** Let  $z_0 = (\text{least } z)[g_j(z, \text{val}(s)) = 0]$ . Since  $\chi(r, s) = 0$  we have

$$z_0 < v(r) < v(SUCC(r)).$$

Now, if  $z_0 < z$  and  $g_j(z_0, x) = 0$  then  $g_j(z, x) = 0$ . Thus,

$$g_j(v(SUCC(r)), s) = 0$$

and the claim follows.

By the claim, the subterm of  $t_1$  reduces to 0, and  $t_1$  reduces by (1) to

$$H_j(SUCC(r), s)$$

which is strongly terminating by induction on the list.

**Induction step** We suppose as the induction hypothesis that for any  $r, s$ , and  $u$  that are strongly terminating and  $\chi(r, s) = d, t = T_j(r, s, u)$  is strongly terminating.

Suppose  $\chi(r, s) = d + 1$ .

**Case a**  $v(u) = 0$ . Then the argument is as for case (a) of the basis of this Case 2.

**Case b**  $v(u) \neq 0$ . Then the equation used is (2) and

$$t_1 = T_j(SUCC(r), s, G_j(SUCC(r), s)).$$

All subterms are strongly terminating and we note that

$$\chi(SUCC(r), s) = d$$

by definition of  $\chi$ . Thus, by the induction hypothesis on  $\chi(r, s)$  we have that  $t_1$  and (hence)  $t$  are strongly terminating.

This concludes the case 2.

**Case 3**  $\lambda$  is  $F_j$  and  $t = F_j(s_1, \dots, s_k)$ .

The only equation that applies is (3) which means that

$$t_1 = T_j(0, s_1, \dots, s_k, SUCC(0))$$

which is strongly terminating by case 3. ■

Up to case 2, where  $T_j$  appears, our proof could be replaced by a shorter but more advanced argument based on the recursive path ordering.

The tighter bounds for the number of hidden functions in Theorem 3.1.5 are obtained by adapting the construction of  $h, g$ , and  $t$  to include the index  $e$  as an argument. This leads to a new smaller  $R_0$  and a specification

$(\Sigma_0, E_0)$  in which there is only one set of hidden functions and one set of equations obtained from the Kleene  $T$ -predicate. We leave this adaptation of the proof as an exercise.

### 3.6 Concluding remarks

The First Completeness Theorem shows that any computable algebra  $A$  has a finite equational specification that is a complete term rewriting system, but that the number of equations is not bounded. We will show that this latter feature is unavoidable.

Here is an example of a family of finite algebras such that all equational specifications that are complete term rewriting systems are dependent on the size of the signature.

Let  $n > 1$  and  $\Sigma_n$  be the signature

<b>signature</b>	<i>finite set with constants</i>
<b>sort</b>	$s$
<b>constants</b>	$c_1, \dots, c_n : \rightarrow s$
<b>end</b>	

and consider the  $\Sigma_n$ -algebra

$$A_n = (\{a\}; a, \dots, a)$$

wherein the  $n$  constants of  $\Sigma_n$  are identified.

**Theorem 3.6.1.** *Let  $(\Sigma, E)$  be a finite equational TRS such that  $\Sigma_n \subseteq \Sigma$ ,  $(\Sigma, E)$  is complete, and*

$$T(\Sigma, E) \upharpoonright_{\Sigma_n} \cong A_n$$

*Then  $|E| > n - 1$ .*

**Proof.** Let  $t \in T(\Sigma)$  be the normal form of  $c_1$ . Then for  $1 \leq j \leq n$ ,  $t$  is the normal form of  $c_j$ . To see this note that

$$A_n \models c_1 = c_j$$

and, because  $A_n$  is specified by  $(\Sigma, E)$ ,

$$E \vdash c_1 = c_j$$

so  $c_1$  and  $c_j$  have a common reduct as  $(\Sigma, E)$  is confluent. Thus we deduce that *at most one* of  $c_1, \dots, c_n$  could be the normal form  $t$ .

Suppose that  $c_j$  is not the normal form  $t$ . Then there must be a rule

$$e_j \equiv l(X) = r(X)$$

that applies to  $c_j$  for  $X = X_1, \dots, X_l$ . Thus,  $l(X) = X_i$  or  $l(X) = c_j$ .

If  $l(X) \equiv X_i$  then the TRS  $(\Sigma, E)$  cannot be terminating and indeed according to the definitions  $(\Sigma, E)$  does not qualify as a TRS, so we exclude this case and assume that  $l(X) = c_j$ .

Now for the  $n - 1$  cases where  $c_j$  is not a normal form we obtain  $n - 1$  equations  $e_j$  with LHS  $c_j$ . Thus  $|E| > n - 1$ . ■

Finally, if we consider semicomputable algebras then, in contrast to the theorem for computable algebras, the following characterisation, using infinitely many equations but not using hidden functions, can be proved:

**Theorem 3.6.2 (Completeness Theorem for Semicomputable Algebras).** *Let  $A$  be a minimal  $\Sigma$ -algebra. Then the following are equivalent:*

1.  *$A$  is semicomputable.*
2. *There is an infinite recursively enumerable equational specification  $(\Sigma, E)$  such that*
  - (a)  *$(\Sigma, E)$  is a complete term rewriting system;*
  - (b)  *$T(\Sigma, E) \cong A$ .*

*However, the recursively enumerable complete term rewriting specification  $(\Sigma, E)$  cannot be replaced by a recursive complete term rewriting specification, nor can it be replaced by a recursively enumerable orthogonal complete term rewriting specification.*

These two results appear in Bergstra and Tucker [1993d].

## 4 Domains and approximations for topological algebras

Suppose we want to compute on an uncountable structure such as the field of real numbers  $\mathbb{R}$ , or some other algebra from the list in Section 1.3. Then, by cardinality considerations, there is obviously no possibility of  $\mathbb{R}$  being a computable structure. In fact, the elements of  $\mathbb{R}$  are in general truly infinite objects (Cauchy sequences or Dedekind cuts) with no finite descriptions. However, we can compute on  $\mathbb{R}$  by computing on concrete *approximations* of elements of  $\mathbb{R}$ . The result of a computation is then interpreted or defined as the ‘limit’ of the computed approximations, where such a limit exists. This is the technique used in recursive analysis, see e.g. Turing [1936], Rice [1954], Grzegorzczuk [1955; 1957], Goodstein [1961], Aberth [1980], and Pour-El and Richards [1989].

Is there a general method or a general class of structures which captures computations via approximations as in the example of the reals  $\mathbb{R}$ ? We shall show that the class of *structured Scott-Ershov domains* does capture an appropriate notion of approximation and that these can be used to represent very large classes of topological algebras in the sense of Section 1.5. Thus we obtain a uniform theory of approximations for these topological algebras via the approximations of the representing structured domain.

Furthermore, when the representing structured domain is effective then the representation induces a notion of effectivity on the topological algebra.

In Section 4.1 we begin by discussing a notion of approximation and approximation structure, primarily in order to motivate our choice of structured Scott–Ershov domains as a class of representing structures for topological algebras. Domains are briefly reviewed in Section 4.2 together with a precise definition of *domain representability*. Then, in Section 4.3, we show that topological algebras obtained through certain inverse limit constructions are domain representable in a strong sense. These include all ultrametric algebras with non-expansive operations. In order to achieve domain representability for more algebras, such as the real numbers  $\mathbb{R}$ , we need to introduce a notion of an element in a domain being ‘total’ or ‘large’, which is done in Section 4.4. Using this notion of totality we show in Section 4.5 that every locally compact Hausdorff algebra is domain representable. By similar techniques one obtains that every metric algebra is domain representable.

## 4.1 Approximation structures and topologies

What kind of objects are approximations and how do they relate to the elements they approximate? Ideally we seek of approximations that they are, in some sense, ‘concrete’ or ‘finite’ relative to the elements they approximate. In addition we require that each element is completely determined by its set of approximations. Thus two elements are distinct precisely when there is an approximation of one element that is not an approximation of the other. Finally we want each element to be the ‘limit’ of its set of approximations. This enables us to compute on the set of approximations and then to transfer these computations to the set of approximated elements using the limit process.

Our goal is to obtain a uniform treatment of the theory of approximations which covers most algebras that we are interested in. This is met by considering a particular class of structures as representations for algebras, as described in Section 1.5. The choice of representations we consider is the class of structured Scott–Ershov domains, defined in Section 4.2. The purpose of the present section is to set the scene for our choice of representations.

A Scott–Ershov domain models a very satisfying approximation theory. The set of approximations for a domain  $D$  is its set of compact elements  $D_c$ . A compact element in a domain is indeed concrete or finite in a precise technical sense. Each element  $x$  in  $D$  is completely determined by its set of compact approximations  $\text{approx}(x)$  and  $x$  is the limit of  $\text{approx}(x)$ . Hence a domain does satisfy our requirements listed above. Moreover, domains have further nice properties. The set of approximations  $D_c$  of  $D$  is part of the domain  $D$  itself; that is, each approximation of an element in  $D$  is itself in  $D$ . We say that the approximations of elements in  $D$  are *explicit* in



$D$ . This allows for a simple theory of effective approximations on domains. Furthermore, the approximations of a domain  $D$  correspond precisely to the approximations obtained when considering  $D$  as a topological space with the Scott topology, and this is reflected exactly by the ordering of the domain. Finally, there is a pragmatic motivation for our choice of representations. Large classes of topological algebras have representations which are structured Scott–Ershov domains and these are obtained in a canonical manner. Given a topological algebra  $A$  with an appropriate approximation structure  $P$ , we obtain the representing structured Scott–Ershov domain as the ideal completion of  $P$ . In particular, we see that various types of completion processes can be dressed up as ideal completions.

We will now discuss a general notion of approximation with the intent of supporting our choice of representations.

Let us consider the reals  $\mathbb{R}$ . A first reasonable attempt is to say that the rational numbers  $\mathbb{Q}$  make up the concrete real numbers and that each real number is approximated by rational numbers. Although true, it does not follow that we should regard  $\mathbb{Q}$  as *the set of approximations for*  $\mathbb{R}$ . For example, do 3 and 3.14 both approximate  $\pi$ ? If yes, then certainly 3.14 is a better approximation of  $\pi$  than 3. However, it is not reasonable to say that 3.14 is a *better approximation* than 3; that is, when the ‘of  $\pi$ ’ is deleted. In fact, in a sense to be made precise below, we claim that a rational or real number can only approximate itself. What is needed, and what is missing in the example of  $\mathbb{Q}$  in  $\mathbb{R}$ , is that an approximation structure should contain information on how well elements are approximated. For  $\mathbb{R}$  we will choose the approximations to be the set of closed intervals  $[a, b]$  with rational endpoints  $a \leq b$  with the intended meaning that

$[a, b]$  approximates a real number  $r$  if  $r \in [a, b]$ .

Each interval  $[a, b]$  does implicitly contain a measure in the following sense: whenever  $[a, b]$  approximates  $r_1$  and  $r_2$  then  $|r_1 - r_2| \leq b - a$ . Note that each approximation  $[a, b]$  is ‘finite’ in the sense that it is a compact interval, and compactness may be regarded as a finiteness property. In addition the representation of an approximation  $[a, b]$  is finite since representations of rational numbers are finite.

Let us consider the problem of approximation abstractly. Suppose that  $A$  is a set. To say that a set  $P$  is an approximation for  $A$  should mean that elements of  $P$  are approximations for, or approximate, elements of  $A$ . That is, there is a relation, the *approximation relation*, from  $P$  to  $A$ . At this stage we pay no attention to the particular nature of the elements of  $P$ . However, we do require that  $P$  completely determines  $A$  in a non-trivial way. This leads us to the following definition.

**Definition 4.1.1.** Let  $A$  be a set. A set  $P$  along with a relation  $\prec$  from  $P$  to  $A$  is said to be an *approximation for*  $A$  if

1.  $(\forall x \in A)(\exists p \in P)(p \prec x)$ , and
2.  $(\forall x, y \in A)(x = y \Leftrightarrow \{p \in P : p \prec x\} = \{p \in P : p \prec y\})$ .

It follows that the set  $A$  can be identified with a family of non-empty subsets of its approximation set  $P$ . Often one can characterise these subsets or, at least, say that they have certain properties. In fact, they are often ideals of some kind.

Let  $(P, \prec)$  be an approximation for  $A$ . Then  $\prec$  induces a relation  $\sqsubseteq$  on  $P$ , called the *refinement order* obtained from or induced by  $\prec$ , in a natural way: for  $p, q \in P$  let

$$p \sqsubseteq q \Leftrightarrow (\forall x \in A)(q \prec x \Rightarrow p \prec x). \quad (*)$$

Thus  $p \sqsubseteq q$  expresses that  $q$  is a better approximation than  $p$ , or  $q$  refines  $p$ , in the sense that  $q$  approximates fewer elements in  $A$  than does  $p$ . The following observation is immediate from the definition.

**Proposition 4.1.2.** *Let  $(P, \prec)$  be an approximation for  $A$ . Then the structure  $P = (P; \sqsubseteq)$  is a preorder (i.e.  $\sqsubseteq$  is reflexive and transitive), where  $\sqsubseteq$  is the refinement order obtained from  $\prec$ .*

Thus we see that a set  $P$  of approximations has a natural structure which is compatible with the approximation relation  $\prec$  via the connection  $(*)$ . In practice, one often decides on natural relations  $\sqsubseteq$  and  $\prec$  simultaneously, but with the compatibility  $(*)$  in mind. Therefore we extend our definition as follows.

**Definition 4.1.3.** A preorder  $P = (P; \sqsubseteq)$  is an *approximation for  $A$  relative to  $\prec$* , if  $(P, \prec)$  is an approximation for  $A$  such that

$$p \sqsubseteq q \Leftrightarrow (\forall x \in A)(q \prec x \Rightarrow p \prec x).$$

Each set  $A$  has a trivial approximation, viz. the approximation  $(A, =)$ . For a non-trivial example, let us return to the real numbers  $\mathbb{R}$ . We let

$$\mathbb{P} = \{[a, b] : a \leq b, a, b \in \mathbb{Q}\} \cup \{(-\infty, \infty)\}$$

and we order  $\mathbb{P}$  by reverse inclusion, that is

$$[a, b] \sqsubseteq [c, d] \Leftrightarrow [a, b] \supseteq [c, d].$$

The approximation relation  $\prec$  from  $\mathbb{P}$  to  $\mathbb{R}$  is defined by

$$[a, b] \prec x \Leftrightarrow x \in [a, b].$$

Note that  $\mathbb{P} = (\mathbb{P}; \sqsubseteq)$  is a computable structure in the sense of Section 1.4. We have also added a least element  $(-\infty, \infty)$ , even though it is not a

compact set in  $\mathbb{R}$ , mainly as a matter of convenience. The least element contains essentially no information since it approximates all elements. Clearly  $\mathbb{P} = (\mathbb{P}; \sqsubseteq)$  is an approximation for  $\mathbb{R}$  relative to  $\prec$ . Thus when studying the theory of approximations of  $\mathbb{R}$  we consider the structure  $\mathbb{R}$  *together with* the approximation structure  $\mathbb{P} = (\mathbb{P}; \sqsubseteq)$  and the approximation relation  $\prec$  between  $\mathbb{P}$  and  $\mathbb{R}$ .

Our further analysis makes explicit use of topological notions. Indeed, topology may be thought of as a theory of approximation with its primary concern with ‘closeness’ or ‘nearness’, fundamental for concepts such as continuity, limit, and connectedness. All topological concepts we use are standard and can be found in any of the many standard accounts of the subject such as Dugundji [1966] and Kelley [1955] and also in Smyth [1992].

Do the open sets of a topological space form an approximation structure for the space? Let  $X = (X, \tau)$  be a topological space. Let  $\prec$  be the relation from  $\tau$  to  $X$  defined by, for  $U \in \tau$  and  $x \in X$ ,

$$U \prec x \Leftrightarrow x \in U;$$

that is, an open set approximates all its members.

**Proposition 4.1.4.**  *$(\tau, \prec)$  is an approximation for  $X$  if, and only if,  $(X, \tau)$  is a  $T_0$ -space.*

**Proof.** Suppose that  $(X, \tau)$  is a  $T_0$ -space. We need to verify (1) and (2) of Definition 4.1.1. The case of (1) is trivial. For (2) assume that  $x, y \in X$  are distinct. By the  $T_0$  property there is an open set  $U$  containing  $x$  but not  $y$  or there is an open set  $V$  containing  $y$  but not  $x$ . In either case this means that

$$\{U \in \tau : x \in U\} \neq \{U \in \tau : y \in U\}$$

so (2) holds. Similarly, if (2) holds then  $(X, \tau)$  is  $T_0$ . ■

It follows that when  $X = (X, \tau)$  is a  $T_0$ -space then  $\tau = (\tau; \supseteq)$  is an approximation for  $X$  relative to  $\prec$ .

Of course it is unnecessary, and in general contradicting our requirement of having ‘concrete’ approximations, to consider the family of all open sets  $\tau$ . It clearly suffices to consider a topological base  $\mathcal{B}$  for  $\tau$  in order for Proposition 4.1.4 to hold. Thus,  $(\mathcal{B}; \supseteq)$  is an approximation for  $X$  relative to the relation  $\prec$  between  $\mathcal{B}$  and  $X$  and  $(\mathcal{B}; \supseteq)$  contains as much information as does  $(\tau; \supseteq)$ . Thus we normally choose, if possible, a topological base consisting of ‘concrete’ open sets.

Recall the stated fact that the ‘concrete’ approximations in a Scott-Ershov domain are explicit in the domain. Let  $P$  be an approximation for  $A$ . When is it possible to embed  $P$  into  $A$  in a natural way? More precisely, is there a natural order  $\leq$  on  $A$  and an embedding  $\iota : P \rightarrow A$  such that  $(\iota[P], \leq)$  is an approximation for  $A$ ?

In general, an embedding is too much to hope for. However, each approximation  $P$  of  $A$  induces an explicit approximation on  $A$  in a natural way as follows.

Let  $(P, \prec)$  be an approximation for  $A$ . Define a relation  $\leq$  from  $A$  to  $A$  by, for  $x, y \in A$ ,

$$x \leq y \Leftrightarrow (\forall p \in P)(p \prec x \Rightarrow p \prec y);$$

that is, the set of approximations of  $x$  is contained in the set of approximations of  $y$ . Thus  $x$  approximates  $y$  in the sense that everything that can be said about  $x$  also can be said about  $y$ . Note that  $\leq$  is a partial order by 4.1.1 (2). Furthermore,

$$x \leq y \Leftrightarrow (\forall z \in A)(z \leq x \Rightarrow z \leq y).$$

It follows that  $\leq$  is an approximation relation from  $A$  to  $A$  and that  $(A, \leq)$  is an approximation for  $A$  in the precise sense of Definition 4.1.1.

Now consider the refinement relation  $\leq'$  induced on  $A$  by  $\leq$ :

$$x \leq' y \Leftrightarrow (\forall z \in A)(y \leq z \Rightarrow x \leq z).$$

Then, trivially,  $x \leq' y \Leftrightarrow x \leq y$ . Thus the refinement order on  $A$  coincides with the induced approximation relation from  $A$  to  $A$ . We summarise our observations.

**Proposition 4.1.5.** *Let  $(P, \prec)$  be an approximation for  $A$  and let  $\leq$  be the relation on  $A$  defined by*

$$x \leq y \Leftrightarrow (\forall p \in P)(p \prec x \Rightarrow p \prec y).$$

*Then  $\leq$  is an approximation relation from  $A$  to  $A$ . Furthermore,  $\leq$  is a partial order and  $(A; \leq)$  is an approximation for  $A$  relative to  $\leq$ .*

We have shown that each approximation  $P$  for  $A$  induces an approximation on  $A$  where the approximation relation coincides with the refinement order. In this way we have made the approximations explicit in  $A$ . However, it still remains to satisfy the requirement of having concrete approximations, i.e. to have an embedding  $\iota : P \rightarrow A$  such that  $(\iota[P]; \leq)$  is an approximation of  $A$  relative to  $\leq$ .

Consider a  $T_0$  topological space  $X = (X, \tau)$  with its approximation structure  $(\tau; \supseteq)$ . Then, in the literature, the approximation order  $\leq$  induced on  $X$  is called the *specialisation order*.

**Proposition 4.1.6.** *Let  $X = (X, \tau)$  be a topological space which is  $T_0$  and let  $\leq$  be the specialisation order on  $X$ . Then  $\leq$  is the discrete order (i.e.  $\leq$  is  $=$ ) if, and only if,  $X$  is a  $T_1$ -space.*



**Proof.** Suppose  $X$  is  $T_1$ . Then for distinct points  $x, y \in X$  there are open sets  $U$  and  $V$  such that  $x \in U$  and  $y \notin U$  and  $x \notin V$  and  $y \in V$ , so that  $x$  and  $y$  are not related by  $\leq$ . Thus the order  $\leq$  is the discrete one for  $T_1$ -spaces. The converse is similar. ■

It follows that for  $T_1$  spaces  $X = (X, \tau)$ , elements in  $X$  only approximate themselves relative to the approximation order  $\leq$  induced by  $\tau$ . In particular there is no proper subset  $P \subseteq X$  such that  $(P; \leq)$  is an approximation for  $X$  relative to  $\leq$ . Thus, in order to have non-trivial concrete *explicit* approximations induced by the topology, we are forced to consider spaces with very weak separation properties, such as Scott–Ershov domains. As we shall see in the next section, a domain  $D$  has the property that the set of compact elements  $D_c \subseteq D$  consists of ‘finite’ elements and that  $(D_c; \leq)$  is an approximation for  $D$  relative to the specialisation order  $\leq$ . In addition,  $\leq$  coincides with the domain order  $\sqsubseteq$ .

## 4.2 Representing topological algebras by structured domains

In this section we shall make precise what we mean by a topological algebra being represented by a domain. Recall that a topological algebra is an algebra whose carrier set is a topological space and whose operations are continuous. For notation, terminology, and basic theory of domains we refer to Stoltenberg-Hansen *et al.* [1994]. See also Abramsky and Jung [1994] in this Handbook (Volume 3).

Our general strategy or method to capture the effective content of a topological algebra is as follows. Given a topological algebra  $A$  we find, if possible, a computable approximation  $P$  for  $A$ . Then we *complete*  $P$  to obtain a generally larger structure  $D$ , also containing ideal elements, so that  $P$  also is an approximation for  $D$ . In this way the approximation of  $A$  is reflected in  $D$ . The structure  $D$  will be used to represent  $A$ .

We find it convenient and not much of a restriction to confine our attention to the following class of approximation structures.

**Definition 4.2.1.** A partial order  $P = (P; \sqsubseteq, \perp)$  with least element  $\perp$  is a *conditional upper semilattice with least element* (abbreviated *culs*) if whenever  $\{a, b\} \subseteq P$  is consistent in  $P$ , that is, has an upper bound in  $P$ , then  $a \sqcup b$  (the least upper bound or supremum of  $a$  and  $b$ ) exists in  $P$ .

The assumption of a least element is not essential. It does reflect that there is at least one approximation and it is also convenient for the existence of least fixed points. We now briefly describe how to complete a *culs* in order to obtain its corresponding domain.

**Definition 4.2.2.** Let  $P = (P; \sqsubseteq, \perp)$  be a *culs*. Then  $I \subseteq P$  is an *ideal* if

1.  $\perp \in I$ ,
2. if  $a \in I$  and  $b \sqsubseteq a$  then  $b \in I$ , and



3. if  $a, b \in I$  then  $a \sqcup b$  exists in  $P$  and  $a \sqcup b \in I$ .

Let  $P$  be a c usl, let  $a \in P$ , and set  $[a] = \{b \in P : b \sqsubseteq a\}$ . It is easily verified that  $[a]$  is an ideal, precisely because  $P$  is a c usl. The ideal  $[a]$  is called the *principal ideal* generated by  $a$ . It is the smallest ideal containing  $a$ .

**Definition 4.2.3.** Let  $P$  be a c usl. Then let  $\overline{P} = \{I \subseteq P : I \text{ is an ideal}\}$  and define the *ideal completion* of  $P$  to be the structure  $\overline{P} = (\overline{P}; \subseteq, [\perp])$ .

The ideal completion  $\overline{P}$  of a c usl  $P$  is a Scott–Ershov domain. We quickly recall the definitions and establish our notation.

**Definition 4.2.4.**

1. Let  $D = (D; \sqsubseteq)$  be a partially ordered set. A set  $A \subseteq D$  is *directed* if  $A \neq \emptyset$  and whenever  $x, y \in A$  then there is  $z \in A$  such that  $x \sqsubseteq z$  and  $y \sqsubseteq z$ .
2. Let  $D = (D; \sqsubseteq, \perp)$  be a partially ordered set with least element  $\perp$ . Then  $D$  is a *complete partial order* (abbreviated *cpo*) if whenever  $A \subseteq D$  is directed then  $\bigsqcup A$  (the least upper bound or supremum of  $A$ ) exists in  $D$ .
3. Let  $D$  be a cpo. An element  $a \in D$  is said to be *compact* or *finite* if whenever  $A \subseteq D$  is a directed set and  $a \sqsubseteq \bigsqcup A$  then there is  $x \in A$  such that  $a \sqsubseteq x$ . The set of compact elements in  $D$  is denoted by  $D_c$ .
4. A cpo  $D$  is an *algebraic cpo* if for each  $x \in D$ , the set

$$\text{approx}(x) = \{a \in D_c : a \sqsubseteq x\}$$

is directed and  $x = \bigsqcup \text{approx}(x)$ .

5. A cpo  $D$  is a *Scott–Ershov domain*, or simply a *domain*, if  $D$  is an algebraic cpo such that if the set  $\{a, b\} \subseteq D_c$  is consistent, i.e. has an upper bound in  $D$ , then  $a \sqcup b$  exists in  $D$ .

It is not hard to prove that the compact elements  $D_c$  of a domain  $D$  form a c usl. Furthermore, a Scott–Ershov domain is *consistently complete*; that is, every consistent set has a supremum.

In this chapter we fix the meaning of the word *domain*. A domain is a Scott–Ershov domain, that is a consistently complete algebraic cpo.

Domains are exactly ideal completions of c usl's.

**Theorem 4.2.5 (Representation theorem).**

1. Let  $P$  be a c usl. Then the ideal completion  $\overline{P} = (\overline{P}; \subseteq, [\perp])$  is a domain. Furthermore,  $\overline{P}_c = \{[a] : a \in P\}$ . Finally, the map  $\iota : P \rightarrow \overline{P}_c$  defined by  $\iota(a) = [a]$  is an order-preserving bijection.
2. Let  $D = (D; \sqsubseteq, \perp)$  be a domain. Then  $\overline{D}_c \cong D$ , where  $\overline{D}_c$  is the ideal completion of the c usl  $D_c$  and where the isomorphism is witnessed by an order-preserving bijection.

The appropriate topology on domains is known as the Scott topology.

**Definition 4.2.6.** Let  $D = (D; \sqsubseteq, \perp)$  be a domain. The *Scott topology* on  $D$  is given by:  $U \subseteq D$  is open if

1.  $x \in U$  &  $x \sqsubseteq y \Rightarrow y \in U$ , and
2.  $x \in U \Rightarrow (\exists a \in \text{approx}(x))(a \in U)$ .

Condition (1) is often referred to as the *Alexandrov condition* and condition (2) as the *Scott condition*. Note that the sets

$$B_a = \{z \in D : a \sqsubseteq z\} \text{ for } a \in D_c$$

are open and form a topological base for the Scott topology. These sets are compact, in fact in a strong sense: every open cover of  $B_a$  has a subcover consisting of *one* open set. This provides us with a technical reason for saying that the basic open sets are ‘concrete’ or ‘finite’ and hence that the compact elements of a domain are ‘finite’.

When considering a domain  $D$  as a topological space we assume that  $D$  is given the Scott topology. Recall the specialisation order on topological spaces defined before Proposition 4.1.6.

We have the following easy, but for our purpose fundamental observation. It says that the topological approximation for  $D$  via basic open sets is embeddable in  $D$  and that the approximation order is the domain order. This observation is a main motivation for attempting to represent topological algebras by domains.

**Proposition 4.2.7.** Let  $D = (D; \sqsubseteq, \perp)$  be a domain. Then the specialisation order on  $D$  coincides with the domain order  $\sqsubseteq$ .

**Proof.** Let  $\leq$  be the specialisation order on  $D$  with respect to the Scott topology. Suppose  $x \sqsubseteq y$  in  $D$ . Then for each open set  $V$  we have, by the Alexandrov condition of Definition 4.2.6,  $x \in V \Rightarrow y \in V$ , that is  $x \leq y$ . Conversely, suppose  $x \not\sqsubseteq y$ . Then there is  $a \in \text{approx}(x) - \text{approx}(y)$ . But then for the basic open set  $B_a = \{z \in D : a \sqsubseteq z\}$  we have  $x \in B_a$  while  $y \notin B_a$ , that is  $x \not\leq y$ . ■

It follows that  $(D_c; \sqsubseteq)$  is an approximation for  $D$  relative to  $\sqsubseteq$  in the sense of Definition 4.1.3.

A function  $f : D \rightarrow E$  between domains is usually said to be *continuous* if  $f$  is monotone and preserves suprema of directed sets, that is

$$\begin{aligned} x \sqsubseteq y &\Rightarrow f(x) \sqsubseteq f(y), \text{ and} \\ f(\bigsqcup A) &= \bigsqcup f[A], \text{ for each directed set } A \subseteq D. \end{aligned}$$

These conditions correspond to the Alexandrov and Scott conditions, respectively, in the definition of the Scott topology. It easily follows that  $f : D \rightarrow E$  is continuous in the above order-theoretic sense if, and only if,

$f$  is continuous in the topological sense with respect to the Scott topologies. We recall some further easy facts about continuity.

**Proposition 4.2.8.** *Let  $D$  and  $E$  be domains.*

1. *A function  $f : D \rightarrow E$  is continuous if, and only if, for each  $x \in D$ ,  $f(x) = \bigcup \{f(a) : a \in \text{approx}(x)\}$ .*
2. *Each monotone function  $f : D_c \rightarrow E$  has a unique continuous extension  $\bar{f} : D \rightarrow E$ .*

In order to represent a topological algebra, we must represent the operations of the algebra, that is we need to consider domains as algebras.

**Definition 4.2.9.** A structure  $D = (D; \sqsubseteq, \perp; x_1, \dots, x_p, \Psi_1, \dots, \Psi_q)$  is a *structured domain* or  $\Sigma$ -domain for a signature  $\Sigma$  if

1.  $(D; \sqsubseteq, \perp)$  is a domain,
2. each  $x_i \in D$ , where  $p$  is given by  $\Sigma$ , and
3. each  $\Psi_j$  is a continuous  $n_j$ -ary operation on  $D$ , that is  $\Psi_j : D^{n_j} \rightarrow D$  is continuous, where  $D^{n_j}$  is given the product topology, and  $q$  and the arities  $n_j$  are given by  $\Sigma$ .

We aim to represent a topological  $\Sigma$ -algebra  $A$  using a  $\Sigma$ -domain  $D$ . However, the ‘concrete’ or ‘finite’ approximations are explicit in  $D$ , which is a main reason we chose to consider domains, while this is rarely the case for  $A$ . Therefore we choose, if possible, a  $\Sigma$ -substructure  $D_A \subseteq D$  consisting of ‘large’ or ‘total’ elements of  $D$  to represent  $A$ .

By  $D_A$  being a  $\Sigma$ -substructure of  $D$  we mean that  $D_A$  is a substructure of  $D$  with respect to the constants and operations named by  $\Sigma$ .

**Definition 4.2.10.** A topological  $\Sigma$ -algebra  $A = (A; a_1, \dots, a_p, \sigma_1, \dots, \sigma_q)$  is *representable* by a  $\Sigma$ -domain  $D = (D; \sqsubseteq, \perp; \hat{a}_1, \dots, \hat{a}_p, \hat{\sigma}_1, \dots, \hat{\sigma}_q)$  if there is a  $\Sigma$ -substructure  $D_A = (D_A; \hat{a}_1, \dots, \hat{a}_p, \hat{\sigma}_1, \dots, \hat{\sigma}_q)$  of  $D$  and a  $\Sigma$ -epimorphism

$$v_A : D_A \rightarrow A$$

which is continuous with respect to the subspace topology of  $D_A$ . The triple  $(D, D_A, v_A)$  is a *domain representation* of  $A$ .

We say that a topological  $\Sigma$ -algebra  $A$  is *domain representable* if  $A$  is representable by some  $\Sigma$ -domain  $D$ .

Most often the  $\Sigma$ -domain  $D$  will be constructed by first choosing an approximation  $\text{csl } P$  for  $A$  and then letting the domain be  $D = \bar{P}$ , the ideal completion of  $P$ . This must be done with some care in order to have all operations of  $A$  represented in  $D$ . In addition we need some notion of ‘largeness’ or ‘totality’ of elements in a domain to reflect that elements in  $A$  are (usually) total in the sense that they only approximate themselves. In Section 4.3 we identify totality with maximality. This is sufficient to show that many topological algebras important in computer science, e.g.

all ultrametric algebras with non-expansive operations, are domain representable. To reach further, to metric algebras and locally compact Hausdorff algebras, we need a more refined notion of totality. This is discussed in Sections 4.4 and 4.5.

We close by showing that domain representable algebras are closed under some usual standard constructions.

The Cartesian product of topological  $\Sigma$ -algebras  $A$  and  $B$  is the  $\Sigma$ -algebra  $A \times B$ , where the operations are defined coordinatewise, and  $A \times B$  is given the product topology. It is standard to show that  $A \times B$  is a topological  $\Sigma$ -algebra.

Recall that the Cartesian product  $D \times E$  of domains  $D$  and  $E$  is given by

$$D \times E = (D \times E; \sqsubseteq, (\perp_D, \perp_E))$$

where  $(x, y) \sqsubseteq (z, w) \Leftrightarrow x \sqsubseteq_D z \ \& \ y \sqsubseteq_E w$ . The Scott topology on  $D \times E$  is the product topology of the Scott topologies on  $D$  and  $E$ . When  $D$  and  $E$  are  $\Sigma$ -domains we define the constants and operations in  $D \times E$  coordinatewise, just as in the case of  $\Sigma$ -algebras.

**Proposition 4.2.11.** *Suppose that the topological  $\Sigma$ -algebras  $A$  and  $B$  are representable by  $\Sigma$ -domains  $D$  and  $E$  respectively. Then the topological  $\Sigma$ -algebra  $A \times B$  is representable by the  $\Sigma$ -domain  $D \times E$ .*

**Proof.** Let  $(D, D_A, v_A)$  and  $(E, E_B, v_B)$  be domain representations of  $A$  and  $B$  respectively. Then  $D_A \times E_B$  is a  $\Sigma$ -substructure of  $D \times E$ . Define  $v_A \times v_B : D_A \times E_B \rightarrow A \times B$  by

$$v_A \times v_B(x, y) = (v_A(x), v_B(y)).$$

It is routine to verify that  $v_A \times v_B$  is a continuous  $\Sigma$ -epimorphism and that  $(D \times E, D_A \times E_B, v_A \times v_B)$  is a domain representation of  $A \times B$ . ■

A subset  $B$  of a topological  $\Sigma$ -algebra  $A$  is a *topological subalgebra* of  $A$  if  $B$  is a subalgebra of  $A$  and  $B$  is a topological algebra with the subspace topology. Note that each subalgebra  $B$  of  $A$  is a topological subalgebra.

**Proposition 4.2.12.** *Suppose the topological  $\Sigma$ -algebra  $A$  is representable by the  $\Sigma$ -domain  $D$ . Then*

1. *each topological subalgebra  $B$  of  $A$  is representable by  $D$ , and*
2. *each continuous homomorphic image  $B$  of  $A$  is representable by  $D$ .*

**Proof.** Let  $(D, D_A, v)$  be a domain representation of  $A$ . For (1) we let  $D_B = v^{-1}[B]$ . Then

$$D_B = (D_B; \hat{a}_1, \dots, \hat{a}_p, \hat{\sigma}_1, \dots, \hat{\sigma}_q)$$

is a  $\Sigma$ -substructure of  $D_A$  since  $v$  is a homomorphism and  $B$  is a  $\Sigma$ -subalgebra of  $A$ . Clearly  $(D, D_B, v|_{D_B})$  is a domain representation of



*B.* For (2) suppose  $\phi : A \rightarrow B$  is a continuous epimorphism. Then  $\phi \circ v : D_A \rightarrow B$  is continuous and  $(D, D_A, \phi \circ v)$  is a domain representation of  $B$ . ■

### 4.3 Inverse limits and ultrametric algebras

In this section we consider topological algebras  $A$  with domain representations  $(D, D_A, v_A)$ , where the set  $D_A$  is contained in the set of maximal elements  $D_m$ . Thus, here, we identify 'large' or 'total' with maximal. The algebras we consider are obtained through certain inverse limit constructions. They include all ultrametric algebras with non-expansive operations.

Inverse limits of algebras occur widely in mathematics and computer science: for example, in constructing power series rings, the p-adic numbers, complete local rings, algebras of infinite trees and terms, infinite words, algebras of infinite streams, algebras of infinite processes in concurrent process theory, and non-well-founded sets and processes. The inverse limit construction is a completion process. The ring of formal power series is the completion of the polynomial ring, algebras of infinite trees and terms are completions of algebras of finite trees and terms, and so on. We shall see how the completion process of the inverse limit construction in topological terms often corresponds to the ideal completion of a cusp.

Throughout our discussion we fix a signature  $\Sigma$ . Let  $I = (I; \leq)$  be a directed set and suppose  $A_i$  is a  $\Sigma$ -algebra for each  $i \in I$ . Further, for each  $i \leq j \in I$  let  $\phi_i^j : A_j \rightarrow A_i$  be a  $\Sigma$ -homomorphism. Then

$$\{A_i : i \in I\}, \{\phi_i^j : i \leq j \in I\}$$

is said to be an *inverse system* of  $\Sigma$ -algebras in case the following two conditions hold for each  $i$  and each  $i \leq j \leq k$ :

1.  $\phi_i^i = \text{id}_{A_i}$ , the identity on  $A_i$ , and
2.  $\phi_i^j \circ \phi_j^k = \phi_i^k$ .

An inverse system is said to be *surjective* if each  $\phi_i^j$  is surjective.

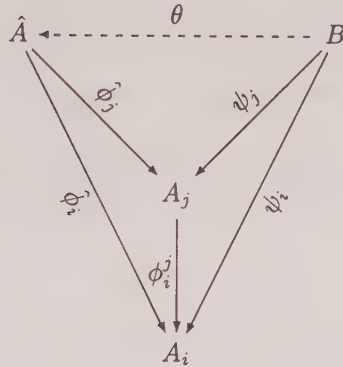
The *projective* or *inverse limit* of the inverse system  $\{A_i : i \in I\}, \{\phi_i^j : i \leq j \in I\}$ , if it exists, is a  $\Sigma$ -algebra  $\hat{A}$  together with a family of  $\Sigma$ -homomorphisms  $\hat{\phi}_i : \hat{A} \rightarrow A_i$  such that for each  $i \leq j \in I$ ,

$$\hat{\phi}_i = \phi_i^j \circ \hat{\phi}_j,$$

and which is a solution to the following universal problem. If  $B$  is a  $\Sigma$ -algebra and  $\psi_i : B \rightarrow A_i$  is a family of  $\Sigma$ -homomorphisms for  $i \in I$  such that for each  $i \leq j \in I, \psi_i = \phi_i^j \circ \psi_j$ , then there is a unique  $\Sigma$ -homomorphism  $\theta$  making the diagrams below commute.

The inverse limit of the inverse system  $\{A_i : i \in I\}, \{\phi_i^j : i \leq j \in I\}$  is often denoted by





$$\varprojlim A_i, \{\hat{\phi}_i : i \in I\},$$

or simply by  $\varprojlim A_i$  leaving the  $\hat{\phi}_i$  implicit. By the usual argument involving a solution to a universal problem, the inverse limit, if it exists, is unique up to a  $\Sigma$ -isomorphism.

In this chapter we only consider certain inverse limits obtained from a family of *separating congruences* on an algebra  $A$ . This will guarantee the existence of the inverse limit. Although a special case of the inverse limit construction it does include a large class of natural examples and it is amenable to domain representability.

Let  $A = (A; a_1, \dots, a_p, \sigma_1, \dots, \sigma_q)$  be a  $\Sigma$ -algebra. A binary relation  $\equiv$  on  $A$  is said to be a *congruence relation* on  $A$  if it is an equivalence relation and if for each operation  $\sigma$  in  $A$ , say  $n$ -ary, if  $x_i \equiv y_i$  for  $i = 1, \dots, n$ , then  $\sigma(x_1, \dots, x_n) \equiv \sigma(y_1, \dots, y_n)$ .

**Definition 4.3.1.** Let  $A = (A; a_1, \dots, a_p, \sigma_1, \dots, \sigma_q)$  be a  $\Sigma$ -algebra and let  $I = (I; \leq, 0)$  be a directed set with least element 0. Then  $\{\equiv_i\}_{i \in I}$  is a *family of separating congruences* on  $A$  if

1. each  $\equiv_i$  is a congruence relation on  $A$ ,
2.  $j \geq i$  and  $x \equiv_j y \Rightarrow x \equiv_i y$ , and
3.  $\bigcap_{i \in I} \equiv_i = \{(x, x) : x \in A\}$ .

For convenience we always assume  $x \equiv_0 y$  for each  $x, y \in A$ . Of course,  $\omega = (\omega; \leq, 0)$  is a directed set with least element. Recall that we identify the set of natural numbers  $\mathbb{N}$  with the ordinal  $\omega$ .

### Examples 4.3.2.

1. On the natural numbers  $\mathbb{N}$ , let  $\equiv_n$  be the equivalence relation corresponding to the partition

$$\{0\}, \{1\}, \{2\}, \dots, \{n-1\}, \{n, n+1, n+2, \dots\}.$$

2. Let  $T(\Sigma, X)$  be the term algebra over a signature  $\Sigma$  and a set of variables  $X$ . Then, for  $t, t' \in T(\Sigma, X)$  let  $t \equiv_n t'$  if  $t$  and  $t'$  are identical up to height  $n - 1$ , for  $n \in \omega$ .
3. Let  $R$  be a local commutative ring whose unique maximal ideal is  $\mathfrak{m}$ . Define for  $x, y \in R$  and  $n \in \omega$ ,  $x \equiv_n y \Leftrightarrow x - y \in \mathfrak{m}^n$ . Then, by Krull's Theorem,  $\{\equiv_n\}_{n \in \omega}$  is a family of separating congruences with respect to the ring operations (see Stoltenberg-Hansen and Tucker [1988]).
4. Let  $2^\omega = \{f \mid f : \omega \rightarrow \{0, 1\}\}$ , the Cantor set. Define for  $f, g \in 2^\omega$ ,  $n \in \omega$ ,

$$f \equiv_n g \Leftrightarrow (\forall i < n)(f(i) = g(i)).$$

Then  $\{\equiv_n\}_{n \in \omega}$  is a family of separating congruences on  $2^\omega$ .

5. Suppose we are given an algebra  $A$  and a family of separating congruences  $\{\equiv_i\}_{i \in I}$  on  $A$ . Let  $J$  be a non-empty possibly infinite set and let  $A^J = \{f \mid f : J \rightarrow A\}$ . We make  $A^J$  into a  $\Sigma$ -algebra in the usual way by saying, for  $\sigma \in \Sigma$   $n$ -ary,

$$\sigma(f_1, \dots, f_n)(j) = \sigma_A(f_1(j), \dots, f_n(j)).$$

We define a family of separating congruences on  $A^J$  as follows. Let

$$I^{(J)} = \{s \mid s : J \rightarrow I \text{ and } s(j) = 0 \text{ a.e.}\},$$

where a.e. means everywhere except on a finite set. We order  $I^{(J)}$  by

$$s \leq t \Leftrightarrow (\forall j \in J)(s(j) \leq t(j)).$$

Clearly  $I^{(J)}$  is a directed set with least element the constant 0 function. For  $a, b \in A^J$  and  $s \in I^{(J)}$  define

$$a \equiv_s b \Leftrightarrow (\forall j \in J)(a(j) \equiv_{s(j)} b(j)).$$

It is easily shown that the family  $\{\equiv_s\}_{s \in I^{(J)}}$  is a family of separating congruences on  $A^J$  over  $I^{(J)}$ .

#### 4.3.1 Construction of the inverse limit

Consider a fixed  $\Sigma$ -algebra  $A = (A; a_1, \dots, a_p, \sigma_1, \dots, \sigma_q)$  together with a family of separating congruences  $\{\equiv_i\}_{i \in I}$ . We shall complete  $A$  using the inverse limit construction.

For  $a \in A$ , let  $[a]_i$  denote the equivalence class of  $\equiv_i$  containing  $a$ , and let

$$A_i = A / \equiv_i = \{[a]_i : a \in A, i \in I\}.$$

Then each  $A_i$  is also a  $\Sigma$ -algebra since  $\equiv_i$  is a congruence relation. For each  $i \leq j \in I$  we define  $\phi_i^j : A_j \rightarrow A_i$  by  $\phi_i^j([a]_j) = [a]_i$ . Then  $\phi_i^j$  is

well-defined by condition (2) of Definition 4.3.1. Furthermore  $\phi_i^i = \text{id}_{A_i}$  and  $\phi_i^j \circ \phi_j^k = \phi_i^k$ . These observations say that  $\{A_i : i \in I\}, \{\phi_i^j : i \leq j \in I\}$  is an *inverse system* which is *surjective*. Let  $\hat{A} = \varprojlim A_i$  be the inverse limit with the associated maps  $\hat{\phi}_i : \hat{A} \rightarrow A_i$ . To verify the existence of  $\hat{A}$  let

$$\hat{A} = \{(a_i)_i \in \prod_{i \in I} A_i : \phi_i^j(a_j) = a_i \text{ for } i \leq j \in I\}.$$

Note that  $\hat{A} \neq \emptyset$  since  $A \neq \emptyset$ . To make  $\hat{A}$  into a  $\Sigma$ -algebra, define for each constant  $a$  in  $A$ ,  $\hat{a} = ([a]_i)_i \in \hat{A}$ , and define for  $\sigma$  a  $t$ -ary operation in  $A$ ,

$$\hat{\sigma}((a_{1,i})_i, \dots, (a_{t,i})_i) = (\sigma(a_{1,i}, \dots, a_{t,i}))_i.$$

It is easily verified that  $\hat{\sigma}$  is well-defined on  $\hat{A}$ . Thus we have obtained the  $\Sigma$ -algebra

$$\hat{A} = (\hat{A}; \hat{a}_1, \dots, \hat{a}_p, \hat{\sigma}_1, \dots, \hat{\sigma}_q).$$

Define  $\hat{\phi}_j : \hat{A} \rightarrow A_j$  by  $\hat{\phi}_j((a_i)_i) = a_j$ . Then each  $\hat{\phi}_j$  is a  $\Sigma$ -homomorphism, and  $\hat{\phi}_i = \hat{\phi}_i^j \circ \hat{\phi}_j$  whenever  $i \leq j$ .

To show that  $\hat{A}$  together with the family of  $\Sigma$ -homomorphisms  $\hat{\phi}_j$  is a solution to the universal problem above, suppose first that the  $\Sigma$ -homomorphism  $\theta$  exists for  $B$  and  $\psi_i$ . Then, for each  $b \in B$  and for each  $i$ ,  $\hat{\phi}_i(\theta(b)) = \psi_i(b)$  so that  $\theta(b) = (\psi_i(b))_i$ . This shows that  $\theta$  is unique. To show the existence of  $\theta$  we need to verify that  $\theta : B \rightarrow \hat{A}$  defined by  $\theta(b) = (\psi_i(b))_i$  is a well-defined  $\Sigma$ -homomorphism. First of all, for  $i \leq j$ ,  $\hat{\phi}_i^j(\psi_j(b)) = \psi_i(b)$  so  $(\psi_i(b))_i \in \hat{A}$ . Furthermore, for an  $m$ -ary operation  $\sigma$ ,

$$\begin{aligned} \theta(\sigma_B(b_1, \dots, b_m)) &= (\psi_i(\sigma_B(b_1, \dots, b_m)))_i \\ &= (\sigma_{A_i}(\psi_i(b_1), \dots, \psi_i(b_m)))_i \\ &= \hat{\sigma}((\psi_i(b_1))_i, \dots, (\psi_i(b_m))_i) \\ &= \hat{\sigma}(\theta(b_1), \dots, \theta(b_m)) \end{aligned}$$

which says that  $\theta$  is a  $\Sigma$ -homomorphism.

We define  $\Sigma$ -homomorphisms  $v_i : A \rightarrow A_i$ , for each  $i \in I$ , by

$$v_i(a) = [a]_i.$$

Then, for  $i \leq j \in I$ ,  $\phi_i^j(v_j(a)) = v_i(a)$  so there is a unique  $\Sigma$ -embedding  $\theta : A \rightarrow \hat{A}$ .

Observe that the disjoint union  $\mathcal{C} = \dot{\bigcup} \{A_i : i \in I\}$  is an approximation for  $\hat{A}$ , in the sense of Definition 4.1.1, with respect to the approximation relation  $\prec$  defined by

$$[a]_i \prec x \Leftrightarrow \hat{\phi}_i(x) = [a]_i.$$

This gives rise to a natural topology on  $\hat{A}$  generated by the topological base

$$\hat{B}(a, i) = \{x \in \hat{A} : \hat{\phi}_i(x) = [a]_i\} \text{ for } a \in A \text{ and } i \in I,$$

making  $\hat{A}$  into a topological algebra. Similarly, the family of separating congruences  $\{\equiv_i\}_{i \in I}$  provides a natural topology on  $A$  generated by the topological base

$$B(a, i) = \{b \in A : a \equiv_i b\} \text{ for } a \in A \text{ and } i \in I,$$

making  $A$  into a topological algebra. It is easily seen that the unique embedding  $\theta : A \rightarrow \hat{A}$ , provided by the inverse limit construction, is continuous. In fact,  $\theta$  is a homeomorphism between  $A$  and its image  $\theta[A]$ .

**Examples 4.3.3.** Referring to Examples 4.3.2. we have that the completion of  $\mathbb{N}$  as in (1) is the one-point compactification, where one element  $\infty$  is added. The completion of  $T(\Sigma, X)$  in (2) is the set  $T^\infty(\Sigma, X)$  of all finite and infinite terms. The completion of a local ring  $R$  as in (3) is the standard construction of completion for local rings. Finally, the completion of the Cantor space gives us the Cantor space back, i.e. the embedding  $\theta$  of the Cantor space is a bijection, since the Cantor space already is complete.

Given an algebra  $A$  and a family of separating congruences  $\{\equiv_i\}_{i \in I}$  satisfying some properties described below, we will construct a structured domain  $D(A)$  such that  $\hat{A} = \varprojlim A_i$  is embedded into  $D(A)$ , in fact into the set of maximal elements  $D(A)_m$ , and such that  $D(A)_c$  consists of the equivalence classes  $[a]_i$  for  $a \in A$  and  $i \in I$ . In other words, the approximations of the inverse limit are precisely the finite or compact elements in  $D(A)$ .

**Definition 4.3.4.** A family of separating congruences  $\{\equiv_i\}_{i \in I}$  on  $A$  is *upward consistent* if

$$a \equiv_i b \ \& \ a \equiv_j b \Rightarrow (\exists k \geq i, j)(a \equiv_k b).$$

In order to obtain consistent completeness for  $D(A)$  we need in addition to the above to assume that the index set  $I$  is closed under finite suprema. In that case upward consistency is characterised by

$$a \equiv_i b \ \& \ a \equiv_j b \Rightarrow (a \equiv_{i \vee j} b),$$

where  $\vee$  is the supremum operation on  $I$ .

Clearly  $\omega = (\omega; \leq, 0)$  is closed under finite suprema and  $\{\equiv_n\}_{n \in \omega}$  is upward consistent. Consider Example 4.3.2 (5). If  $I$  is closed under finite suprema and finite infima, i.e.  $I$  is a lattice, and  $\{\equiv_i\}_{i \in I}$  is upward consistent then also  $I^{(J)}$  is closed under finite suprema and finite infima and  $\{\equiv_s\}_{s \in I^{(J)}}$  on  $A^J$  is upward consistent.

### 4.3.2 Construction of the domain $D(A)$

Let  $A = (A; a_1, \dots, a_p, \sigma_1, \dots, \sigma_q)$  be a  $\Sigma$ -algebra, let  $\{\equiv_i\}_{i \in I}$  be a fixed family of separating congruences on  $A$  which is upward consistent, and assume that the index set  $I$  is closed under finite suprema. We start by representing the carrier set of  $A$ , with the topology induced by the family of separating congruences, in a domain  $D(A)$ . Let  $A_i = A / \equiv_i$  be the set of equivalence classes of  $\equiv_i$  and let

$$\mathcal{C} = \bigcup \{A_i : i \in I\},$$

the disjoint union of the  $A_i$ . We denote an element of  $\mathcal{C}$  by  $[a]_i$  where  $a \in A$  and  $i \in I$ , thus letting the  $i$  indicate that  $[a]_i$  is taken from  $A_i$  in the disjoint union. We define a partial order on  $\mathcal{C}$  by

$$[a]_i \sqsubseteq [b]_j \Leftrightarrow i \leq j \text{ and } a \equiv_i b.$$

That is,  $[a]_i \sqsubseteq [b]_j$  if and only if  $i \leq j$  and  $\phi_i^j([b]_j) = [a]_i$ . Thus  $\sqsubseteq$  is the refinement order on  $\mathcal{C}$  induced by the approximation relation  $\prec$  from  $\mathcal{C}$  to  $\hat{A}$ , defined above for inverse limits.

We claim that  $\mathcal{C}$  with this ordering is a csl. Clearly  $[a]_0$  is the least element in  $\mathcal{C}$  by our standing assumption on  $\equiv_0$  that all elements are  $\equiv_0$  equivalent, where 0 is the least element in  $I$ . Now suppose  $[a]_i$  and  $[b]_j$  are bounded in  $\mathcal{C}$  by, say,  $[c]_k$ . Then  $i, j \leq k$  and  $a \equiv_i c$  and  $b \equiv_j c$ . Thus  $[c]_{i \vee j}$  is an upper bound. To see that  $[c]_{i \vee j}$  is the least upper bound let  $[d]_m$  be another upper bound. This means that  $i, j \leq m$  and  $a \equiv_i d$  and  $b \equiv_j d$ . But then  $c \equiv_i d$  and  $c \equiv_j d$  and hence, by upward consistency,  $c \equiv_{i \vee j} d$ . It follows that  $[c]_{i \vee j} \leq [d]_m$ .

Having shown that  $\mathcal{C}$  is a csl we now define  $D(A) = \bar{\mathcal{C}}$ , the ideal completion of the csl  $\mathcal{C}$ . Identifying the principal ideals in  $D(A)$  with their generating elements in  $\mathcal{C}$ , we have  $D(A)_c = \mathcal{C}$ .

Let  $(a_i)_i \in \hat{A}$ . Then the consistency property of the sequence  $(a_i)_i$ , and the fact that  $I$  is directed, assures that the set  $\{a_i : i \in I\} \subseteq \mathcal{C}$  is directed. Thus we may define  $\Psi : \hat{A} \rightarrow D(A)$  by

$$\Psi((a_i)_i) = \bigsqcup \{a_i : i \in I\}.$$

Furthermore, the set  $\{a_i : i \in I\}$  is closed downwards in  $\mathcal{C}$  so that  $\text{approx}(\Psi((a_i)_i)) = \{a_i : i \in I\}$ . It follows that  $\Psi$  is injective. It is easy to see that if  $J \subseteq \mathcal{C}$  is an ideal such that  $(\forall i \in I)(\exists a \in A)([a]_i \in J)$  then  $J$  is maximal in  $D(A)$ . It follows that  $\Psi[\hat{A}] \subseteq D(A)_m$ . We summarise our observations.

**Theorem 4.3.5.** *Let  $A$  be a  $\Sigma$ -algebra and let  $I$  be a directed set with a least element and closed under finite suprema. Suppose  $\{\equiv_i\}_{i \in I}$  is a family of separating congruences on  $A$  which is upward consistent and let*



$\hat{A} = \varprojlim A / \equiv_i$ . Then the following hold.

1.  $D(A)$  is a domain with  $D(A)_c = \{[a]_i : a \in A, i \in I\}$ .
2. There is an embedding  $\Psi : \hat{A} \rightarrow D(A)$  such that for  $(a_i)_i \in \hat{A}$ ,  $\text{approx}(\Psi((a_i)_i)) = \{a_i : i \in I\}$  and  $\Psi[\hat{A}] \subseteq D(A)_m$ .
3. The embedding  $\Psi$  is a homeomorphism between  $\hat{A}$  and  $\Psi[\hat{A}]$  when  $\hat{A}$  is given the topology induced by  $\{\equiv_i\}_{i \in I}$  and  $\Psi[\hat{A}]$  is given the subspace topology of the Scott topology on  $D(A)$ .

**Proof.** It only remains to prove (3), which follows from the easy fact that for  $a \in A, i \in I$ , and  $x \in \hat{A}$ ,

$$[a]_i \subseteq \Psi(x) \Leftrightarrow x \in \hat{B}(a, i).$$

### Examples 4.3.6.

1. Let  $I = \{0, 1\}$  with  $0 < 1$ . Let  $A$  be a set and define  $a \equiv_1 b \Leftrightarrow a = b$ . Then  $\{\equiv_i\}_{i \in I}$  is a family of separating congruences on  $A$  which is upward consistent. It is not hard to see that  $D(A) \cong A_\perp$ , the flat domain over  $A$ .
2. Let  $\{\equiv_n\}_{n \in \omega}$  be a family of separating congruences on the set  $A$ . Then

$$D(A) = \hat{A} \dot{\cup} \left( \bigcup \{A / \equiv_n : n \in \omega\} \right),$$

and  $D(A)_c = \bigcup \{A / \equiv_n : n \in \omega\}$ . Note that  $\bigcup \{A / \equiv_n : n \in \omega\}$  is an  $\omega$ -tree and that the infinite branches correspond to the elements of  $\hat{A}$ , that is  $D(A)_m = \hat{A}$ .

3. Let  $\{\equiv_s\}_{s \in I^{(J)}}$  be a family of separating congruences on  $A^J$  as defined in 4.3.2 (5). Then  $D(A^J)_c$  need no longer be a tree even in the case  $I = \omega$ .

To establish notation for the remaining part of this section we let  $\Theta : A \rightarrow D(A)$  be defined by

$$\Theta(a) = \Psi(\theta(a))$$

where  $\theta : A \rightarrow \hat{A}$  is the unique embedding provided by the inverse limit construction and  $\Psi : \hat{A} \rightarrow D(A)$  is the embedding of Theorem 4.3.5. It is easily shown that  $\Theta$  is continuous and, in fact, a homeomorphism between  $A$  and  $\Theta[A]$ .

We have yet to represent the algebra  $\hat{A} = (\hat{A}; \hat{a}_1, \dots, \hat{a}_p, \hat{\sigma}_1, \dots, \hat{\sigma}_q)$  by a structured domain; that is, we have to show how to interpret the elements  $\hat{a}_i$

and the operations  $\hat{\sigma}_j$ . Thus we need to find elements  $w_1, \dots, w_p \in D(A)_m$  and continuous operations  $\phi_1, \dots, \phi_q$  on  $D(A)$  so that

$$D(A) = (D(A); \sqsubseteq, \perp; w_1, \dots, w_p, \phi_1, \dots, \phi_q)$$

is a  $\Sigma$ -domain,  $\Psi[\hat{A}]$  is a  $\Sigma$ -subalgebra of  $D(A)$ , and  $(D(A), \Psi[\hat{A}], \Psi^{-1})$  is a domain representation of  $\hat{A}$ .

For the constants we set

$$w_i = \Psi(\hat{a}_i) \text{ for } i = 1, \dots, p.$$

For the operations it clearly suffices to choose continuous  $\phi_j$  such that for each  $x_1, \dots, x_{n_j} \in \hat{A}$ ,

$$\Psi(\hat{\sigma}_j(x_1, \dots, x_{n_j})) = \phi_j(\Psi(x_1), \dots, \Psi(x_{n_j})) \text{ for } j = 1, \dots, q. \quad (*)$$

We say that  $\phi_j$  is a *faithful* representation of  $\hat{\sigma}_j$  when  $(*)$  holds.

The choice of the  $\phi_j$  is by no means unique. There is usually some freedom in how to define the operations on compact elements, and this is crucial for certain fixed point arguments.

**Definition 4.3.7.** Let  $A$  be a  $\Sigma$ -algebra with a family of separating congruences  $\{\equiv_i\}_{i \in I}$  which is upward consistent and where the index set  $I$  is closed under finite suprema. Let  $\lambda : I^n \rightarrow I$  be a function which is monotone in each argument. We say that a function  $f : A^n \rightarrow A$  is  $\lambda$ -congruent if for each  $i_1, \dots, i_n \in I$ ,

$$a_1 \equiv_{i_1} b_1, \dots, a_n \equiv_{i_n} b_n \Rightarrow f(a_1, \dots, a_n) \equiv_{\lambda(i_1, \dots, i_n)} f(b_1, \dots, b_n).$$

Let  $f : A^n \rightarrow A$  be  $\lambda$ -congruent. Then define

$$\phi_f^\lambda([a_1]_{i_1}, \dots, [a_n]_{i_n}) = [f(a_1, \dots, a_n)]_{\lambda(i_1, \dots, i_n)}.$$

**Lemma 4.3.8.** Let  $f : A^n \rightarrow A$  be  $\lambda$ -congruent. Then  $\phi_f^\lambda : D(A)_c^n \rightarrow D(A)$  is well-defined and monotone.

**Proof.** That  $\phi_f^\lambda$  is well-defined follows from the  $\lambda$ -congruence of  $f$ . To prove that  $f$  is monotone it suffices to prove monotonicity in each argument. So we assume for notational simplicity that  $f$  is unary. Suppose  $[a]_i \sqsubset [b]_j$ , that is  $i \leq j$  and  $a \equiv_i b$ . By the  $\lambda$ -congruence of  $f$  we have  $f(a) \equiv_{\lambda(i)} f(b)$ , and by the monotonicity of  $\lambda$  we have  $\lambda(i) \leq \lambda(j)$ . Thus

$$\phi_f^\lambda([a]_i) = [f(a)]_{\lambda(i)} = [f(b)]_{\lambda(i)} \sqsubset [f(b)]_{\lambda(j)} = \phi_f^\lambda([b]_j).$$



It follows that we can extend  $\phi_f^\lambda$  uniquely to a continuous function  $\phi_f^\lambda : D(A)^n \rightarrow D(A)$ . This is the *representation* of  $f$  with respect to  $\lambda$  on  $D(A)$ .

**Definition 4.3.9.** Let  $f : A^n \rightarrow A$  be  $\lambda$ -congruent. Then  $f$  is *continuous with respect to  $\lambda$*  if  $\lambda$  is unbounded, that is for each  $i \in I$  there is  $i_1, \dots, i_n \in I$  such that  $i \leq \lambda(i_1, \dots, i_n)$ .

**Lemma 4.3.10.** Let  $f : A^n \rightarrow A$  be continuous with respect to  $\lambda$ . Then  $\phi_f^\lambda$  is a faithful representation of  $f$ , that is for each  $a_1, \dots, a_n \in A$ ,

$$\Theta(f(a_1, \dots, a_n)) = \phi_f^\lambda(\Theta(a_1), \dots, \Theta(a_n)),$$

where  $\Theta : A \rightarrow D(A)$  is the embedding obtained from the inverse limit construction.

**Proof.** For notational simplicity we assume that  $n = 1$ . First recall that  $\Theta(a) = ([a]_i)_{i \in I}$  for  $a \in A$  and hence that  $\Theta(a) = \bigsqcup \{[a]_i : i \in I\}$  and  $\text{approx}(\Theta(a)) = \{[a]_i : i \in I\}$ . Thus

$$\begin{aligned} \phi_f^\lambda(\Theta(a)) &= \phi_f^\lambda(\bigsqcup \{[a]_i : i \in I\}) \\ &= \bigsqcup \{\phi_f^\lambda([a]_i) : i \in I\} \\ &= \bigsqcup \{[f(a)]_{\lambda(i)} : i \in I\} \\ &\sqsubseteq \bigsqcup \{[f(a)]_j : j \in I\} \\ &= \Theta(f(a)). \end{aligned}$$

The converse inequality follows from the continuity condition on  $\lambda$ . For given  $j \in I$  choose  $i \in I$  such that  $\lambda(i) \geq j$ . Then

$$\phi_f^\lambda(\Theta(a)) \supseteq [f(a)]_{\lambda(i)} \supseteq [f(a)]_j$$

so  $\phi_f^\lambda(\Theta(a)) \supseteq \Theta(f(a))$ . ■

We have shown that each function  $f : A^n \rightarrow A$ , continuous with respect to  $\lambda$ , extends continuously and faithfully to the whole domain  $D(A)$  by  $\phi_f^\lambda$ . Suppose that the index set  $I$  is a lattice. Then, assuming  $\sigma_i$  is  $n$ -ary, we define  $\lambda_i : I^n \rightarrow I$  by

$$\lambda_i(i_1, \dots, i_n) = i_1 \wedge \dots \wedge i_n.$$

It follows that  $\sigma_i$  is  $\lambda_i$ -congruent and that  $\lambda_i$  is unbounded. Thus  $\phi_{\sigma_i}^{\lambda_i}$  represents  $\sigma_i$  faithfully on  $D(A)$ . We have arrived at our main theorem of this section.

**Theorem 4.3.11.** Let  $A = (A; a_1, \dots, a_p, \sigma_1, \dots, \sigma_q)$  be a  $\Sigma$ -algebra together with a family of separating congruences  $\{\equiv_i\}_{i \in I}$  on  $A$  which is upward consistent and suppose the index set  $I$  is a lattice with a least element.

Let  $\hat{A} = \varprojlim A / \equiv_i$  and let  $\Theta : A \rightarrow D(A)$  and  $\Psi : \hat{A} \rightarrow D(A)$  be the embeddings defined above. Then  $A$  and  $\hat{A}$  are representable by the  $\Sigma$ -domain

$$D(A) = (D(A); \sqsubseteq, \perp; \Theta(a_1), \dots, \Theta(a_p), \phi_{\sigma_1}^{\lambda_1}, \dots, \phi_{\sigma_q}^{\lambda_q}),$$

where  $(D(A), \Psi[\hat{A}], \Psi^{-1})$  is a domain representation of  $\hat{A}$  and  $(D(A), \Theta[\hat{A}], \Theta^{-1})$  is a domain representation of  $A$ .

**Proof.** It only remains to show that  $\hat{A}$  is represented by  $D(A)$ . But  $\theta[A]$  is dense in  $\hat{A}$  and  $\hat{A}$  is Hausdorff, so each operation  $\hat{\sigma}$  in  $\hat{A}$  is the unique continuous extension (via  $\theta$ ) of the corresponding operation  $\sigma$  in  $A$ . ■

One should observe that the particular choice of  $\lambda$  plays no role in the behaviour of  $\phi_f^\lambda$  on  $\Psi[\hat{A}]$ , as long as  $f$  is continuous with respect to  $\lambda$ . However, as already remarked, the choice of  $\lambda$  does affect the behaviour of  $\phi_f^\lambda$  on  $D(A)_c$  and hence the process of taking fixed points of such functions. For further discussion, and applications to equation solving over inverse limit algebras, see Stoltenberg-Hansen and Tucker [1991; 1993].

It remains to consider briefly ultrametric algebras. We shall show that an algebra  $A$  together with a family of separating congruences  $\{\equiv_n\}_{n \in \omega}$  can be seen as an ultrametric algebra with non-expansive operations. Conversely, from an ultrametric algebra  $A$  with non-expansive operations we obtain a family of separating congruences  $\{\equiv_n\}_{n \in \omega}$  on  $A$  such that  $A$  with the ultrametric is equivalent to  $A$  with the topology induced by  $\{\equiv_n\}_{n \in \omega}$  as topological algebras.

Recall that a metric space  $(X, d)$  is an *ultrametric space*, and  $d$  is an *ultrametric*, if  $d$  satisfies the following stronger form of the triangle inequality:

$$d(x, y) \leq \max\{d(x, z), d(z, y)\}.$$

Consider a  $\Sigma$ -algebra  $A$  together with a family  $\{\equiv_n\}_{n \in \omega}$  of separating congruences on  $A$ . Let  $(r_n)$  be a sequence of strictly decreasing positive real numbers such that  $r_n \rightarrow 0$ . Then define an ultrametric  $d$  on  $A$  by

$$d(x, y) = \begin{cases} 0 & \text{if } x = y \\ r_n & \text{if } x \neq y, \text{ where } n \text{ is least s.t. } x \not\equiv_n y. \end{cases}$$

The distance function  $d$  is defined everywhere by (3) of Definition 4.3.1. It is easily seen to be an ultrametric, the ultrametric property following from (2) of Definition 4.3.1. Similarly we define an ultrametric  $\hat{d}$  on  $\hat{A} = \varprojlim A / \equiv_n$  by

$$\hat{d}(x, y) = \begin{cases} 0 & \text{if } x = y \\ r_n & \text{if } x \neq y, \text{ where } n \text{ is least s.t. } \hat{\phi}_n(x) \neq \hat{\phi}_n(y). \end{cases}$$

The operations in  $A$  and  $\hat{A}$  are continuous with respect to the ultrametrics  $d$  and  $\hat{d}$  respectively. In fact, since  $\equiv_m$  is a congruence relation for each operation  $\sigma$  in  $A$  and each  $m \in \omega$ , the following stronger condition holds:

$$d(\sigma(x_1, \dots, x_n), \sigma(y_1, \dots, y_n)) \leq \max\{d(x_i, y_i) : 1 \leq i \leq n\}. \quad (**)$$

An operation  $\sigma$  satisfying  $(**)$  is said to be *non-expansive*. Note that the topological algebra  $A$  is independent of the choice of the sequence  $(r_n)$ . For purposes of computability, one therefore often chooses a recursive sequence of rational numbers, for example  $r_n = 2^{-n}$ . In fact, this part of the theory need not refer to the real numbers  $\mathbb{R}$  at all.

**Theorem 4.3.12.** *Suppose  $A$  is a  $\Sigma$ -algebra together with a family of separating congruences  $\{\equiv_n\}_{n \in \omega}$  and let  $\hat{A} = \varprojlim A / \equiv_n$ . Let  $(r_n)$  be a sequence of strictly decreasing positive real numbers such that  $r_n \rightarrow 0$  and let  $d$  and  $\hat{d}$  be the ultrametrics for  $A$  and  $\hat{A}$  respectively, as defined above with respect to the sequence  $(r_n)$ . Then the following hold.*

1.  $\hat{A}$  is a complete ultrametric space.
2. The unique  $\Sigma$ -embedding  $\theta : A \rightarrow \hat{A}$  is an isometry with respect to  $d$  and  $\hat{d}$ .
3.  $\theta[A]$  is dense in  $\hat{A}$ .

That is,  $\hat{A}$  is the ultrametric completion of  $A$ .

**Proof.** (1.) To show that  $\hat{A}$  is complete, suppose  $(x_n)$  is a Cauchy sequence in  $\hat{A}$ . Then for each  $n$  there is some  $N_n$  such that  $s, t \geq N_n \Rightarrow \hat{d}(x_s, x_t) < r_n$  or, in other words,  $\hat{\phi}_n(x_s) = \hat{\phi}_n(x_t)$ . Let  $f : \omega \rightarrow \omega$  be the increasing function which given  $n$  chooses the least such  $N_n$ . We claim that  $x = (\hat{\phi}_n(x_{f(n)}))_n \in \hat{A}$ . For, given  $m \leq n$ , we have that

$$\begin{aligned} \phi_m^n(\hat{\phi}_n(x_{f(n)})) &= \hat{\phi}_m(x_{f(n)}) \\ &= \hat{\phi}_m(x_{f(m)}) \end{aligned}$$

since  $f$  is increasing. To see that  $x_n \rightarrow x$  consider  $n$  and  $t \geq f(n)$ . Then

$$\hat{\phi}_n(x) = \hat{\phi}_n(x_{f(n)}) = \hat{\phi}_n(x_t)$$

so that  $\hat{d}(x, x_t) < r_n$ .

(2.) The metrics are defined with respect to the same sequence  $(r_n)$  so  $\theta$  is an isometry since  $v_n = \hat{\phi}_n \circ \theta$  for each  $n$ , where  $v_n : A \rightarrow A / \equiv_n$  is the canonical quotient mapping.

(3.) Given  $x \in A$  and  $n$ , choose  $a \in A$  such that  $v_n(a) = \hat{\phi}_n(x)$ . Then  $\hat{d}(x, \theta(a)) < r_n$  so  $\theta[A]$  is dense in  $A$ . ■



An ultrametric algebra with non-expansive operations can be viewed in a topologically equivalent way as an algebra together with a family of separating congruences  $\{\equiv_n\}_{n \in \omega}$ . Using this method we construct a completion of an ultrametric algebra  $A$  which is topologically equivalent to the metric completion of  $A$ .

**Theorem 4.3.13.** *Suppose  $A$  is an ultrametric  $\Sigma$ -algebra with non-expansive operations. Then there is a complete ultrametric  $\Sigma$ -algebra  $\hat{A}$  with non-expansive operations, and a continuous  $\Sigma$ -embedding  $\theta : A \rightarrow \hat{A}$  such that each  $x \in \hat{A}$  is the limit of some sequence  $(\theta(a_n))$  where  $(a_n)$  is a Cauchy sequence in  $A$ . In particular,  $\theta[A]$  is dense in  $\hat{A}$ .*

**Proof.** We may without loss of topological generality assume that the ultrametric  $d$  on  $A$  is bounded. Define a family of separating congruences  $\{\equiv_n\}_{n \in \omega}$  by

$$x \equiv_n y \Leftrightarrow d(x, y) \leq r_n$$

where  $(r_n)$  is a strictly decreasing sequence of positive real numbers such that  $r_n \rightarrow 0$ . Then we obtain  $\hat{A} = \varprojlim A / \equiv_n$  and the injective  $\Sigma$ -homomorphism  $\theta : A \rightarrow \hat{A}$  by the construction of Theorem 4.3.12. To see that  $\theta$  is continuous, just observe that the identity on  $A$  is a homeomorphism between  $(A, d)$  and  $(A, d')$  where  $d'$  is the ultrametric on  $A$  obtained from  $\{\equiv_n\}_{n \in \omega}$ . This observation also suffices for proving that  $\theta[A]$  is dense in  $\hat{A}$ . ■

**Corollary 4.3.14.** *Each ultrametric  $\Sigma$ -algebra  $A$  with non-expansive operations is domain representable.*

**Proof.** This follows from the construction above and Theorem 4.3.11. ■

One use of the above equivalence is that we can analyse certain results for ultrametric spaces or algebras using ideal completions. For example, the Banach fixed point theorem for complete ultrametric spaces is a consequence of the fixed point theorem for the representing domain. A similar analysis can be made for arbitrary metric spaces.

## 4.4 Total elements in domains

In the previous section we represented topological algebras using maximal elements of a domain. This method is sufficient for many algebras of interest in computer science, but by no means for all. For example, it does not suffice for important structures such as the reals  $\mathbb{R}$ . The reason is the following. It is an elementary fact that the maximal elements in a domain have a clopen topological base, i.e. a topological base where each set is both closed and open. This is by far not the case for  $\mathbb{R}$  nor for many other mathematical structures of interest. Thus we need a new concept of totality in a domain, which is more general than that of maximality, and which can be used to represent more structures in domains. We will use a notion

of totality in domains, due to Berger [1993], which is based on a topological semantics for propositional intuitionistic truth. Notions of totality in qualitative domains and in Scott–Ershov domains have also been studied in Normann [1989; 1990], Kristiansen [1993] and Kristiansen and Normann [1994].

In this section we introduce and develop the basic theory of this notion of totality. Then, in the next section, we apply our theory to show that every locally compact Hausdorff algebra is domain representable. In particular, the algebra  $\mathbb{R} = (\mathbb{R}; a_1, \dots, a_p, \sigma_1, \dots, \sigma_q)$  with continuous operations is a locally compact Hausdorff algebra and hence domain representable. Similar results hold for metric algebras.

We may think of open sets of a topological space as propositions asserting properties of elements of the space. For some topological space  $X$  consider assertions of the form  $x \in U$  where  $x \in X$  and  $U$  is open in  $X$ . Such an assertion is *definitely true* if, in fact,  $x \in U$ . It is *definitely false* if there is another definitely true assertion which contradicts it, that is if there is an open set  $V$  such that  $x \in V$  and  $V \cap U = \emptyset$ . The assertion  $x \in U$  is *decidable* if it is either definitely true or definitely false, that is if  $x \notin \partial U$ , the boundary of  $U$ .

Intuitively, an element  $x \in X$  is total if it is completely determined or described with respect to a sufficiently rich set of decidable propositions, that is if the family of open sets  $U$  for which  $x \notin \partial U$  is sufficiently large. What we mean by sufficiently rich may of course vary depending on the applications we have in mind. For our purposes we need only one notion of sufficiently rich, made precise below. It is easily seen that the notion of totality thus obtained trivialises for Hausdorff spaces in that each element of a Hausdorff space is total. This observation is compatible with the fact that non-trivial approximations only occur in spaces with weak separation properties. Here, we will only be concerned with totality in domains.

To say that an *element*  $x$  in a domain  $D$  is *total* we require that the family of open sets  $U$  for which  $x \in U$  is decidable to be sufficiently rich. To say that a *set*  $M \subseteq D$  is *total* we require that the family of open sets  $U$  for which  $x \in U$  is decidable for *each*  $x \in M$  to be sufficiently rich. Thus we consider two notions: that of a total *element* and that of a total *set*.

**Definition 4.4.1.** Let  $D$  be a domain.

1. A family  $\mathcal{F}$  of open subsets of  $D$  is said to be *separating* if whenever  $x_0, x_1, \dots, x_n \in D$  are topologically separated, that is there are open sets  $U_0, U_1, \dots, U_n$  such that  $x_i \in U_i$  and  $U_0 \cap U_1 \cap \dots \cap U_n = \emptyset$ , then  $x_0, x_1, \dots, x_n$  can be topologically separated using elements of  $\mathcal{F}$ .
2. An element  $x \in D$  *decides* an open set  $U$  if  $x \notin \partial U$ .
3. An element  $x \in D$  *decides* a family  $\mathcal{F}$  of open sets if  $x$  decides each  $U \in \mathcal{F}$ .

4. An element  $x \in D$  is *total* if  $\mathcal{F}(x) = \{U \subseteq D : U \text{ open \& } x \notin \partial U\}$  is a separating family of open sets.
5. A set  $M \subseteq D$  is *total* if  $\mathcal{F}(M) = \{U \subseteq D : U \text{ open \& } (\forall x \in M) (x \notin \partial U)\}$  is a separating family of open sets.

Note that if  $M \subseteq D$  is a total set and  $x \in M$  then  $x$  is a total element. Of course, in order to verify that  $M \subseteq D$  is a total set (or  $x \in D$  is a total element) it suffices to find a family  $\mathcal{F} \subseteq \mathcal{F}(M)$  (or  $\mathcal{F} \subseteq \mathcal{F}(x)$ ) which is separating. Here are some further elementary observations.

**Lemma 4.4.2.** *Let  $D$  be a domain,  $x \in D$ , and let  $U$  be an open subset of  $D$ .*

1. *If  $x$  decides  $U$  and  $x \sqsubseteq y$  then  $y$  decides  $U$ .*
2. *If  $x$  decides  $U$  then there is a  $a \in \text{approx}(x)$  such that  $a$  decides  $U$ .*
3. *If  $\mathcal{F}$  is a finite family of open sets and  $x$  decides  $\mathcal{F}$  then there is a  $a \in \text{approx}(x)$  such that  $a$  decides  $\mathcal{F}$ .*
4. *If  $x \sqsubseteq y$  and  $y \in U$  then  $x \in U \cup \partial U$ .*

**Proof.** Part (1) follows from the Alexandrov condition on open sets and (2) from the Scott condition. Part (3) follows from (1), (2), and the fact that the supremum of finitely many consistent compact elements is compact, and (4) follows, again, from the Alexandrov condition. ■

The following proposition shows that the set of maximal elements  $D_m$  of a domain  $D$  is a total set, so that we do have a generalisation. Recall that, for  $a \in D_c$ ,  $B_a = \{y \in D : a \sqsubseteq y\}$  is a basic open set in the Scott topology.

**Proposition 4.4.3.** *Let  $D$  be a domain. Then  $x \in D_m$  if, and only if,  $x$  decides  $\{B_a : a \in D_c\}$ . In particular,  $D_m$  is a total set.*

**Proof.** Suppose  $x$  is maximal and  $x \notin B_a$  for some  $a \in D_c$ . Then there is  $b \in D_c$  such that  $x \in B_b$  and  $B_a \cap B_b = \emptyset$ , that is  $x \notin \partial B_a$ . Thus  $x$  decides  $\{B_a : a \in D_c\}$ . Conversely, suppose  $x \sqsubset y$ , so that  $x$  is not maximal, and let  $b \in \text{approx}(y) - \text{approx}(x)$ . Then  $y \in B_b$ ,  $x \notin B_b$  and hence  $x \in \partial B_b$  by Lemma 4.4.2 (4), that is  $x$  does not decide  $\{B_a : a \in D_c\}$ . The family  $\{B_a : a \in D_c\}$  is clearly separating since it is a topological base, and hence  $D_m$  is a total set. ■

It follows that the set  $A$  is a total set in the flat domain  $A_\perp$ . Consider the domain  $D = [A_\perp \rightarrow A_\perp]$ . Then it is easy to see that the set of functions in  $D$  taking elements of  $A$  to elements of  $A$ , i.e.  $\{f \in D : f[A] \subseteq A\}$ , is a total set in  $D$ . In other words, identifying  $\perp$  with undefined and total with everywhere defined, the set of total functions is a total set. However, it is not the case that every total function in  $D$  is maximal in  $D$ ; for example a function which takes  $\perp$  to  $\perp$  and which takes a constant value in  $A$  elsewhere. On the other hand, letting  $E = [D \rightarrow A_\perp]$  and saying that

$f \in E$  is total if  $f$  takes total elements in  $D$  to elements in  $A$ , then it is the case that there are maximal elements in  $E$  which are not total. Thus maximality and the natural notion of totality in function spaces are not related.

We remark, and it is routine to verify, that elements  $x_0, x_1, \dots, x_n \in D$  are topologically separated if, and only if, the set  $\{x_0, x_1, \dots, x_n\}$  is inconsistent; that is, does not have an upper bound in  $D$ . Here is a nice characterisation of the total elements in a domain.

**Proposition 4.4.4.** *Let  $D$  be a domain and let  $x \in D$ . Then the following are equivalent.*

1.  $x$  is a total element.
2. Whenever  $x \sqsubseteq y$  and  $x \sqsubseteq z$  then  $y$  and  $z$  are consistent.
3. Whenever  $x_0, \dots, x_k \in D$  are topologically separated then  $x$  and  $x_i$  are topologically separated for some  $i$ .

**Proof.** (1)  $\Rightarrow$  (2). Suppose there are  $y, z \in D$  such that  $x \sqsubseteq y$  and  $x \sqsubseteq z$  but  $y$  and  $z$  are topologically separated. Let  $U$  and  $V$  be open sets separating  $y$  and  $z$ . Then  $x$  does not belong to at least one of  $U$  or  $V$ , since  $U \cap V = \emptyset$ , say  $U$ . But then  $x \in \partial U$  by Lemma 4.4.2 (4), which proves that  $x$  is not total.

(2)  $\Rightarrow$  (3). Let  $x_0, \dots, x_k \in D$  and suppose  $x$  and  $x_i$  are consistent for each  $i$ . Then  $x \sqcup x_0$  and  $x \sqcup x_1$  exists and hence, by (2),  $x \sqcup x_0 \sqcup x_1$  exists. Iterating the argument, using (2), we see that  $x \sqcup x_0 \sqcup \dots \sqcup x_k$  exists. In particular, the set  $\{x_0, \dots, x_k\}$  is consistent.

(3)  $\Rightarrow$  (1). Assume that (3) holds. We prove that  $\mathcal{F}(x)$  is a separating family of open sets. Let  $x_0, \dots, x_k \in D$  be topologically separated. Choose  $a_0 \in \text{approx}(x_0), \dots, a_k \in \text{approx}(x_k)$  so that  $a_0, \dots, a_k$  are topologically separated, that is  $\bigcap_{i=0}^k B_{a_i} = \emptyset$ . By our hypothesis there is  $i$  such that  $x$  and  $a_i$  are topologically separated. It follows that  $x \notin \overline{B_{a_i}}$ , the closure of  $B_{a_i}$ . Let

$$\begin{cases} V_i = B_{a_i} \\ V_j = B_{a_j} \cup (D - \overline{B_{a_i}}) \quad \text{for } j \neq i. \end{cases}$$

Then  $x \in V_j$  for each  $j \neq i$  and  $x \notin \partial V_i$ , so  $x_j \in V_j \in \mathcal{F}(x)$  for  $j = 0, \dots, k$ . To show that  $\bigcap_{j=0}^k V_j = \emptyset$  suppose  $y \in \bigcap_{j=0}^k V_j$ . In particular,  $y \in B_{a_i}$  and hence  $y \in B_{a_j}$  for each  $j$ , contradicting that  $\bigcap_{j=0}^k B_{a_j} = \emptyset$ . ■

A further requirement one might reasonably place on a total set  $M$  is that elements in  $M$  should exist *essentially everywhere* in  $D$ . In the language of topology this means that  $M$  should be dense in  $D$ , that is  $M \cap U \neq \emptyset$  for each non-empty open set  $U$  in  $D$ . Of course, the set of maximal elements of a domain is dense. We state the following fundamental theorem, due to Berger [1993], without proof.



**Theorem 4.4.5.** *Let  $D$  and  $E$  be domains and let  $M \subseteq D$  and  $N \subseteq E$  be total dense sets. Then*

1.  $M \times N$  is a total dense set in  $D \times E$ , and
2.  $\langle M, N \rangle$  is a total dense set in  $[D \rightarrow E]$ , where

$$\langle M, N \rangle = \{f \in [D \rightarrow E] : x \in M \Rightarrow f(x) \in N\}.$$

Without considering the technical notion of totality, it should be clear that  $M \times N$  and  $\langle M, N \rangle$  naturally constitute what should be the total subsets of  $D \times E$  and  $[D \rightarrow E]$  respectively. Thus Theorem 4.4.5 allows us to build natural type structures of total elements. In particular, one can show that the continuous functionals of Kleene [1959] and Kreisel [1959] are representable by domains. (First shown by Ershov [1974; 1977a]. See also Berger [1993], or Stoltenberg-Hansen *et al.* [1994].)

Now we develop some basic general theory. Let  $D$  be a domain and let  $M \subseteq D$  be a set of total elements (so *not* necessarily a total set). Define a binary relation  $\sim$  on  $M$  by

$$x \sim y \Leftrightarrow x \text{ and } y \text{ are consistent in } D.$$

**Lemma 4.4.6.** *The relation  $\sim$  is an equivalence relation on  $M$ .*

**Proof.** The relation  $\sim$  is clearly reflexive and symmetric. To prove transitivity, suppose  $x \sim y$  and  $y \sim z$ . Then  $x \sqcup y$  and  $y \sqcup z$  exist and are extensions of  $y$ . Thus, using the totality of  $y$  and Proposition 4.4.4,  $x \sqcup y$  and  $y \sqcup z$  are consistent and hence  $x \sim z$ . ■

**Definition 4.4.7.** Let  $D$  be a domain and let  $M \subseteq D$  be a set of total elements. Let  $\sim$  be the binary relation on  $M$  defined by  $x \sim y \Leftrightarrow x$  and  $y$  are consistent in  $D$ . Then we say that

$$\tilde{M} = M / \sim = \{[x]_{\sim} : x \in M\}$$

is the *space of total elements*, obtained from  $M \subseteq D$ .

As usual,  $[x]_{\sim}$  denotes the equivalence class with respect to  $\sim$  containing  $x$ . In what follows we shall simply write  $[x]$  for  $[x]_{\sim}$ .

In case the set  $M$  is upwards closed with respect to the domain ordering  $\sqsubseteq$ , we can define a canonical representative  $\tilde{x}$  of  $[x]$ . Of course, if  $x$  is total and  $x \sqsubseteq y$  then  $y$  is also total by Lemma 4.4.2 (1).

**Lemma 4.4.8.** *If  $M \subseteq D$  is a set of total elements which is upwards closed with respect to  $\sqsubseteq$  in  $D$  then each equivalence class  $[x]$  contains a unique element  $\tilde{x}$  maximal in  $D$ .*

**Proof.** We show that the set  $[x]$  is directed. Suppose  $y, z \in [x]$ . Then  $y \sim z$  and hence  $y \sqcup z$  exists in  $M$  by  $y \sim z$ , and hence  $x \sim y \sqcup z$ , that is



$y \sqcup z \in [x]$ . Thus we may set  $\tilde{x} = \sqcup[x]$ . Clearly  $\tilde{x} \in [x]$  and  $\tilde{x}$  is maximal in  $D$ . ■

We say that the element  $\tilde{x}$  is the *canonical (maximal)* representative of  $[x]$ . Unfortunately, there is in general no *continuous* function on  $D$  which given  $x \in M$  has  $\tilde{x}$  as value. This forces us to consider the quotient structure  $\tilde{M}$  rather than the subspace  $\{\tilde{x} : x \in M\}$ .

In order to connect with the topological notion of continuity we consider the following topology on the space of total elements  $\tilde{M}$ . The domain  $D$  is given the Scott topology and  $M$  inherits the subspace topology from  $D$ . Then we give  $\tilde{M}$  the quotient topology from  $M$ . Thus  $U \subseteq \tilde{M}$  is open precisely when  $\bigcup U$  is open in  $M$ , that is if, and only if,  $\bigcup U = V \cap M$  for some Scott open set  $V \subseteq D$ . We call this the *quotient topology* for  $\tilde{M}$ . It is the largest topology making the canonical quotient mapping  $v : M \rightarrow \tilde{M}$  continuous. One can show that if  $M$  is a total set (i.e. not only a set of total elements) then  $\tilde{M}$  will have a clopen topological base. It follows that we cannot represent topological algebras such as  $\mathbb{R}$  using a total set.

We conclude with the easy but important observation that continuous functions between domains induce continuous functions between corresponding spaces of total elements.

**Proposition 4.4.9.** *Let  $D$  and  $E$  be domains and suppose  $M \subseteq D$  and  $N \subseteq E$  are sets of total elements. If  $f : D \rightarrow E$  is a continuous function such that  $f[M] \subseteq N$  then  $\tilde{f} : \tilde{M} \rightarrow \tilde{N}$  defined by  $\tilde{f}([x]) = [f(x)]$  is continuous with respect to the quotient topologies.*

## 4.5 Representability of locally compact Hausdorff algebras

The class of locally compact Hausdorff algebras is a large and natural class of algebras and includes many metric algebras, for example the ring of real numbers  $\mathbb{R}$ . In this section we show that every locally compact Hausdorff algebra is domain representable. Similar techniques can be used to show that each metric algebra is domain representable.

Let us consider a fixed topological space  $X$  which is locally compact and Hausdorff. We first construct a domain  $D$  representing  $X$ . Later we will show how to represent continuous operations on  $X$  as continuous operations on  $D$ .

A first observation is that  $X$  satisfies the following stronger separation property. For a proof we refer to standard texts on topology such as Kelley [1955] or Dugundji [1966].

**Proposition 4.5.1.** *Let  $X$  be a locally compact Hausdorff space. Then  $X$  is regular. In fact, the family of compact neighbourhoods of each point is a base for its neighbourhood system.*

The latter conclusion of the proposition means that for each  $x \in X$  and

each open neighbourhood  $U$  of  $x$  there is a compact set  $F \subseteq U$  such that  $x \in F^O$ , the interior of  $F$ . Recall that each compact subset of a Hausdorff space is closed and that each closed subset of a compact set is compact.

Let  $P'$  be a family of non-empty compact subsets of  $X$  and let  $P = P' \cup \{X\}$ . We order  $P$  by reverse inclusion, that is for  $F, F' \in P$ ,

$$F \subseteq F' \Leftrightarrow F \supseteq F'.$$

Then  $P = (P; \subseteq, X)$  is a partial order with least element  $X$ . We will use  $P$  as an approximation for  $X$ , in the sense of Definition 4.1.3, relative to the approximation order

$$F \text{ approximates } x \Leftrightarrow x \in F.$$

For reasons of computability we want  $P$  as simple as possible, while still being an approximation for  $X$ . What we require is the following:

**Definition 4.5.2.** Let  $X$  be a topological space and let  $P = P' \cup \{X\}$  where  $P'$  is a family of non-empty compact subsets of  $X$ . Then  $P = (P; \subseteq, X)$  is a *culs of compact neighbourhood systems* if the following conditions hold:

1. if  $F, F' \in P$  and  $F \cap F' \neq \emptyset$  then  $F \cap F' \in P$ , and
2. if  $x \in U$ , where  $U$  is open, then  $(\exists F \in P)(x \in F^O \text{ \& } F \subseteq U)$ .

Note that if  $P$  is a culs of compact neighbourhood systems for the locally compact Hausdorff space  $X$  then  $P$  is an approximation for  $X$ . Proposition 4.5.1 tells us that each locally compact Hausdorff space  $X$  has a culs of compact neighbourhood systems, namely  $\mathcal{H}(X) \cup \{X\}$ , where  $\mathcal{H}(X)$  is the set of all non-empty compact sets. For  $\mathbb{R}$ , for example, we can do better by letting

$$\mathbb{P} = \{[a, b] : a \leq b, a, b \in \mathbb{Q}\} \cup \{\mathbb{R}\}.$$

Then  $\mathbb{P}$  is a computable structure.

Now we fix a culs of compact neighbourhood systems  $P$  for the locally compact Hausdorff space  $X$ . Note that, by 4.5.2 (1),  $P$  is a culs and that  $F$  and  $F'$  are consistent if and only if  $F \cap F' \neq \emptyset$ , in which case  $F \sqcup F' = F \cap F'$ .

Let  $\overline{P}$  be the ideal completion of  $P$ . We shall show that  $\overline{P}$  is a domain representing  $X$ . Consider  $I \in \overline{P}$ , that is  $I$  is an ideal over  $P$ . If  $I = \{X\}$  then  $\bigcap I \neq \emptyset$ . Suppose  $F \in I$  where  $F \neq X$  and consider the set  $I_F = \{F' \cap F : F' \in I\}$ . Then  $I_F \subseteq I$ , and  $I$  and  $I_F$  have the finite intersection property by virtue of being directed sets. It follows that

$$\bigcap I = \bigcap I_F \neq \emptyset$$

since  $F$  is compact.

We first aim to show that  $I$  and  $J$  in  $\overline{P}$  are consistent just in case  $(\bigcap I) \cap (\bigcap J) \neq \emptyset$ . We start by showing that this condition is local.

**Proposition 4.5.3.** *Let  $I, J \in \overline{P}$ . Then*

$$(\bigcap I) \cap (\bigcap J) = \emptyset \Leftrightarrow (\exists F \in I)(\exists F' \in J)(F \cap F' = \emptyset).$$

**Proof.** In the non-trivial direction suppose  $I$  and  $J$  are different from  $\{X\}$  and suppose  $F' \cap F'' \neq \emptyset$  for all  $F' \in I$  and  $F'' \in J$ . Choose  $F \in I$  such that  $F \neq X$  and consider  $I_F = \{F' \cap F : F' \in I\}$ . Let

$$\mathcal{F} = \{F' \cap F'' : F' \in I_F \text{ and } F'' \in J\}.$$

Then  $\mathcal{F}$  has the finite intersection property and hence  $\bigcap \mathcal{F} \neq \emptyset$  since  $F$  is compact. It follows that  $(\bigcap I) \cap (\bigcap J) \neq \emptyset$ . ■

Here we pause briefly to consider when a family  $\mathcal{G} \subseteq P$  generates an ideal in  $\overline{P}$ . Clearly  $\mathcal{G}$  must have the finite intersection property since this is true for each ideal. This is also sufficient. For suppose  $\mathcal{G}$  does have the finite intersection property. Let  $\mathcal{G}'$  be defined by

$$\mathcal{G}' = \{G_1 \cap G_2 \cap \dots \cap G_n : G_i \in \mathcal{G} \text{ and } n \geq 1\},$$

and let

$$I_{\mathcal{G}} = \{H \in P : (\exists G' \in \mathcal{G}')(G' \subseteq H)\}.$$

Then clearly  $I_{\mathcal{G}}$  is the smallest ideal containing  $\mathcal{G}$ . We have shown that:

**Proposition 4.5.4.** *A family  $\mathcal{G} \subseteq P$  generates an ideal over  $P$  if, and only if,  $\mathcal{G}$  has the finite intersection property.*

Now we characterise the consistency relation in  $\overline{P}$ .

**Proposition 4.5.5.** *Ideals  $I, J \in \overline{P}$  are consistent in  $\overline{P}$  if, and only if,  $(\bigcap I) \cap (\bigcap J) \neq \emptyset$ .*

**Proof.** Suppose  $I, J \subseteq K \in \overline{P}$ . Then  $\emptyset \neq \bigcap K \subseteq (\bigcap I) \cap (\bigcap J)$ . Conversely, if  $(\bigcap I) \cap (\bigcap J) \neq \emptyset$  then, by Propositions 4.5.3 and 4.5.4, the family

$$\mathcal{K} = \{F \cap F' : F \in I \text{ and } F' \in J\}$$

generates an ideal  $K$ . It is clearly the case that  $I, J \subseteq K$ . ■

Next we characterise the total elements in  $\overline{P}$ . The notion of a total element in a domain is defined in 4.4.1. For  $F \in P$  we let  $[F]$  denote the principal ideal generated by  $F$ . Clearly, for  $I \in \overline{P}$ ,

$$[F] \subseteq I \Leftrightarrow F \in I.$$

Thus the basic open sets of  $\overline{P}$  in the Scott topology are of the form

$$B_F = \{I \in \overline{P} : F \in I\}$$

for  $F \in P$ . It follows from Proposition 4.5.3 that

$$B_F \cap B_G = \emptyset \Leftrightarrow F \cap G = \emptyset.$$

**Proposition 4.5.6.** *An ideal  $I \in \overline{P}$  is a total element in  $\overline{P}$  if, and only if,  $\bigcap I$  is a singleton set.*

**Proof.** Suppose  $x, y \in \bigcap I$  and  $x \neq y$ . Let  $F, G \in P$  be such that  $x \in F, y \in G$  and  $F \cap G = \emptyset$ . Such  $F$  and  $G$  exist since  $X$  is Hausdorff and  $P$  satisfies 4.5.2 (2). By Proposition 4.5.5, the ideal  $I$  has extensions  $I \sqcup [F]$  and  $I \sqcup [G]$ , which must be inconsistent since  $F \cap G = \emptyset$ . It follows, by Proposition 4.4.4, that  $I$  is not total.

Conversely, suppose  $I$  is not total. Then there are  $F, G \in P$  such that  $I \sqcup [F]$  and  $I \sqcup [G]$  exist and are inconsistent. It follows, again from Proposition 4.4.4, that  $F \cap G = \emptyset$ . Choose  $x \in \bigcap(I \sqcup [F])$  and  $y \in \bigcap(I \sqcup [G])$ . Then  $x, y \in \bigcap I$  and  $x \neq y$ . ■

We need to note the following useful fact.

**Lemma 4.5.7.** *Let  $I \in \overline{P}$  and suppose  $V \subseteq X$  is open. If  $\bigcap I \subseteq V$  then there is  $F \in I$  such that  $F \subseteq V$ .*

**Proof.** Suppose not. Then for each  $F \in I, F \cap V^c \neq \emptyset$ , where  $V^c$  is the complement of  $V$  in  $X$ . Clearly  $I \neq \{X\}$ . Choose  $F \in I$  such that  $F \neq X$  and consider the set

$$\mathcal{F} = \{V^c \cap F \cap H : H \in I\}$$

which is a family of closed subsets of the compact set  $F$ . Furthermore,  $\mathcal{F}$  has the finite intersection property since  $I$  is an ideal. It follows, by the compactness of  $F$ , that  $\bigcap \mathcal{F} \neq \emptyset$ . But clearly

$$\bigcap \mathcal{F} \subseteq V^c \cap \left(\bigcap I\right) = \emptyset$$

which provides us with a desired contradiction. ■

Consider the domain  $\overline{P} = (\overline{P}; \subseteq, [X])$  and its set of total elements

$$\overline{P}_t = \{I \in \overline{P} : \bigcap I = \text{singleton}\}.$$

For each  $x \in X$  let  $I_x$  be the ideal generated by the set  $\{F \in P : x \in F^O\}$ .

**Lemma 4.5.8.**

1.  $\bigcap I_x = \{x\}$ .
2. If  $J \in \bar{P}$  and  $\bigcap J = \{x\}$  then  $I_x \subseteq J$ .

**Proof.** (1.) Clearly  $x \in \bigcap I_x$ . Let  $y \in X$  be such that  $x \neq y$ . Choose an open set  $U$  containing  $x$  such that  $y \notin U$ . Then, by condition (2) of Definition 4.5.2, there is  $F \in I_x$  such that  $y \notin F$ .

(2.) Let  $F \in P$  be such that  $x \in F^O$ . Then by Lemma 4.5.7, there is  $G \in J$  such that  $G \subseteq F^O \subseteq F$ . Thus  $F \in J$ , that is  $I_x \subseteq J$ . ■

We know that for each ideal  $I \in \bar{P}_t$  there is  $x \in X$  such that  $\bigcap I = \{x\}$ , and, conversely, for each  $x \in X$  there is an ideal  $I \in \bar{P}_t$  such that  $\bigcap I = \{x\}$ . In fact, there is a least such, namely  $I_x$ . By Proposition 4.4.8 there is also a largest such ideal which we denote by  $I^x$ .

Denote by  $\tilde{P}$ , the quotient of  $\bar{P}_t$  by  $\sim$ , that is

$$\tilde{P} = \bar{P}_t / \sim.$$

Recall that the topology on  $\tilde{P}$  which we consider is the quotient topology obtained from the subspace topology on  $\bar{P}_t$ , which in turn is obtained from the Scott topology on  $\bar{P}$ .

Define a function  $\Phi : \tilde{P} \rightarrow X$  by

$$\Phi([I]_{\sim}) = x \Leftrightarrow \bigcap I = \{x\}.$$

The function  $\Phi$  is clearly a well-defined bijection where the surjectivity is witnessed by the ideals  $I_x$ . We are going to show that  $\Phi$  is a homeomorphism.

**Lemma 4.5.9.** *The function  $\Phi : \tilde{P} \rightarrow X$  is continuous.*

**Proof.** Let  $V \subseteq X$  be an open set and let  $\bar{V} = \{I \in \bar{P} : \bigcap I \subseteq V\}$ . We claim that  $\bigcup \Phi^{-1}[V] = \bar{P}_t \cap \bar{V}$ . If  $I \in \bigcup \Phi^{-1}[V]$  then  $I \in \bar{P}_t$  and  $\bigcap I \subseteq V$ , that is  $I \in \bar{P}_t \cap \bar{V}$ . Conversely, if  $I \in \bar{P}_t \cap \bar{V}$  then  $\bigcap I = \{x\}$  for some  $x \in V$  so  $\Phi([I]_{\sim}) = x \in V$ , that is  $I \in \bigcup \Phi^{-1}[V]$ .

To show that  $\Phi$  is continuous it remains to show that  $\bar{V}$  is open in  $\bar{P}$ . Let  $I \in \bar{V}$  and choose  $F_I \in I$  such that  $F_I \subseteq V$ , by Lemma 4.5.7. Thus  $I \in B_{F_I}$ . Suppose  $J \in B_{F_I}$ . Then  $F_I \in J$  so  $\bigcap J \subseteq F_I \subseteq V$ . It follows that  $B_{F_I} \subseteq \bar{V}$  and hence  $\bar{V}$  is open. ■

Before showing that  $\Phi$  is an open mapping we make the following observation. Let  $\tilde{V} \subseteq \tilde{P}$  be an open set and let  $W = \Phi[\tilde{V}]$ . Since  $\tilde{V}$  is open we have that  $\bigcup \tilde{V} = \bar{V} \cap \bar{P}_t$  where  $\bar{V}$  is open in  $\bar{P}$ . Let  $\bar{V} = \bigcup_{i \in Q} B_{F_i}$ .

**Lemma 4.5.10.**  $F_i \subseteq W$  for each  $i \in Q$ .



**Proof.** Let  $i \in Q$  and let  $x \in F_i$ . Consider the largest ideal  $I^x$  such that  $\bigcap I^x = \{x\}$ . Then  $F_i \in I^x$  so  $I^x \in B_{F_i}$ . Thus  $I^x \in \bigcup \tilde{V}$  so  $[I^x]_{\sim} \in \tilde{V}$  and hence  $x = \Phi([I^x]_{\sim}) \in W$ . ■

We have now arrived at the first part of our main result of representing topological algebras.

**Theorem 4.5.11.** *Let  $X$  be a locally compact Hausdorff space. Suppose  $P$  is a cusp of compact neighbourhood systems for  $X$ . Let  $\bar{P}$  be the ideal completion of  $P$  and let  $\bar{P}_t = \{I \in \bar{P} : \bigcap I = \text{singleton}\}$ . Then  $\bar{P}_t$  is the set of total elements of  $\bar{P}$ . Furthermore,  $\tilde{P} = \bar{P}_t / \sim$  and  $X$  are homeomorphic topological spaces.*

**Proof.** It remains to show that the mapping  $\Phi$  is open. Let  $\tilde{V}$  and  $W$  be as above. Thus

$$\bigcup \tilde{V} = \bar{P}_t \cap \left( \bigcup_{i \in Q} B_{F_i} \right).$$

Let  $x \in W$ . Then  $I_x \in \bigcup \tilde{V}$  and hence  $I_x \in B_{F_i}$  for some  $i \in Q$ . But that means that  $F_i \in I_x$  and hence that  $x \in F_i^O \subseteq F_i \subseteq W$ . This shows that  $W$  is open. ■

**Corollary 4.5.12.** *Let  $X$  be a locally compact Hausdorff space and suppose  $P$  is a cusp of compact neighbourhood systems for  $X$ . Then  $(\bar{P}, \bar{P}_t, \Phi \circ v)$  is a domain representation of  $X$ , where  $v : \bar{P}_t \rightarrow \tilde{P}$  is the canonical quotient mapping and  $\Phi : \tilde{P} \rightarrow X$  is the homeomorphism given above.*

**Proof.** The quotient mapping  $v$  is continuous. ■

It remains for us to show how to represent the operations of a locally compact Hausdorff algebra as continuous operations on the representing domain. We consider a slightly more general situation.

Let  $X$  and  $Y$  be locally compact Hausdorff spaces and let  $P$  and  $Q$  be cusps of compact neighbourhood systems for  $X$  and  $Y$  respectively. Thus  $\tilde{P} = \bar{P}_t / \sim$  and  $\tilde{Q} = \bar{Q}_t / \sim$  are homeomorphic copies of  $X$  and  $Y$ , respectively.

Suppose  $\bar{f} : \bar{P} \rightarrow \bar{Q}$  is a continuous function such that  $\bar{f}[\bar{P}_t] \subseteq \bar{Q}_t$ . Then, by Proposition 4.4.9,  $\bar{f}$  induces a continuous function  $f : X \rightarrow Y$ . We will show that every continuous function from  $X$  into  $Y$  is obtained in this way.

**Definition 4.5.13.** Let  $D$  and  $E$  be domains representing topological spaces  $X$  and  $Y$  respectively, using sets of total elements  $M \subseteq D$  and  $N \subseteq E$ . Let  $\tilde{M} = M / \sim$  and  $\tilde{N} = N / \sim$ , and let  $\Phi : \tilde{M} \rightarrow X$  and  $\Psi : \tilde{N} \rightarrow Y$  be homeomorphisms. Then a continuous function  $f : X \rightarrow Y$

is said to be *represented* by a continuous function  $\bar{f} : D \rightarrow E$  if  $\bar{f}[M] \subseteq N$  and if

$$f(x) = \Psi(\bar{f}(\Phi^{-1}(x)))$$

where  $\tilde{f} : \tilde{M} \rightarrow \tilde{N}$  is defined by  $\tilde{f}([x]_{\sim}) = [\bar{f}(x)]_{\sim}$ . We have the following diagram.

$$\begin{array}{ccc}
 D & \xrightarrow{\bar{f}} & E \\
 \uparrow \iota_M & & \uparrow \iota_N \\
 M & \xrightarrow{\bar{f}|_M} & N \\
 \downarrow v_{\sim} & & \downarrow v_{\sim} \\
 \tilde{M} & \xrightarrow{\tilde{f}} & \tilde{N} \\
 \downarrow \Phi & & \downarrow \Psi \\
 X & \xrightarrow{f} & Y
 \end{array}$$

**Lemma 4.5.14.** *Let  $X$  and  $Y$  be locally compact Hausdorff spaces with cul's of compact neighbourhood systems  $P$  and  $Q$ , respectively. Suppose  $f : X \rightarrow Y$  is continuous. Then there is a continuous function  $\bar{f} : \bar{P} \rightarrow \bar{Q}$  representing  $f$ .*

**Proof.** Define  $\bar{f} : P \rightarrow \bar{Q}$  by  $\bar{f}(X) = \{Y\}$  and for  $F \neq X$ ,

$$\bar{f}(F) = \{G \in Q : f[F] \subseteq G^O\}.$$

It is easily verified that  $\bar{f}(F)$  is an ideal and that  $\bar{f}$  is monotone. We also denote by  $\bar{f}$  its unique continuous extension to all of  $\bar{P}$ . In fact, for  $I \in \bar{P}$ ,  $I \neq \{X\}$ ,

$$\bar{f}(I) = \{G \in Q : (\exists F \in I)(f[F] \subseteq G^O)\}.$$

Let  $x \in X$  and consider the ideal  $I_x = \{F \in P : x \in F^O\}$ . We claim that  $\bigcap \bar{f}(I_x) = \{f(x)\}$ . Clearly  $f(x) \in \bigcap \bar{f}(I_x)$ . To show the converse inclusion, suppose  $w \in Y$  and  $w \neq f(x)$ . Choose an open set  $U$  such that  $f(x) \in U$  but  $w \notin U$ . By the assumptions on  $Q$  there is  $H \in Q$  such that

$$f(x) \in H^O \subseteq H \subseteq U.$$

The set  $f^{-1}[H^O]$  is open in  $X$  by the continuity of  $f$ . Let  $F \in P$  be such that

$$x \in F^O \subseteq F \subseteq f^{-1}[H^O].$$

Thus  $F \in I_x$  and  $f[F] \subseteq H^O \subseteq U$ . In particular,  $H \in \bar{f}(I_x)$  and  $w \notin H$ . We conclude that  $\bigcap \bar{f}(I_x) = \{f(x)\}$ .

The claim shows that  $\bar{f}[\bar{P}_t] \subseteq \bar{Q}_t$ . For suppose  $J \in \bar{P}_t$ . Then  $I_x \subseteq J$  for some  $x \in X$  and hence  $\bar{f}(J) \in \bar{Q}_t$  since  $\bar{f}(I_x) \in \bar{Q}_t$ , by Proposition 4.5.6. Furthermore, recalling the homeomorphisms  $\tilde{P} \cong X$  and  $\tilde{Q} \cong Y$ , the claim gives that  $\bar{f}$  represents  $f$ . ■

We summarise our result.

**Theorem 4.5.15.** *Let  $X$  and  $Y$  be locally compact Hausdorff spaces. Suppose  $P$  and  $Q$  are csl's of compact neighbourhood systems for  $X$  and  $Y$  respectively. Then each continuous function  $\bar{f} : \bar{P} \rightarrow \bar{Q}$  such that  $\bar{f}[\bar{P}_t] \subseteq \bar{Q}_t$  induces a continuous function  $f : X \rightarrow Y$ . Conversely, each continuous function  $f : X \rightarrow Y$  is represented by a continuous function  $\bar{f} : \bar{P} \rightarrow \bar{Q}$ .*

Here is our representation theorem.

**Theorem 4.5.16.** *Let  $A = (A; a_1, \dots, a_p, \sigma_1, \dots, \sigma_q)$  be a topological  $\Sigma$ -algebra which is locally compact and Hausdorff. Then  $A$  is domain representable.*

**Proof.** By Theorem 4.5.11 there is a domain  $D$  representing the topological space  $A$ . Hence, by Proposition 4.2.11,  $D^n$  represents  $A^n$ . Let  $\sigma$  be an  $n$ -ary operation of  $A$ , that is  $\sigma : A^n \rightarrow A$  and  $\sigma$  is continuous. Then, by Theorem 4.5.15, the operation  $\sigma$  is represented by some continuous  $\bar{\sigma} : D^n \rightarrow D$ . Thus the  $\Sigma$ -domain  $D = (D; \sqsubseteq, \perp; w_1, \dots, w_p, \bar{\sigma}_1, \dots, \bar{\sigma}_q)$  represents  $A$ , where  $w_1, \dots, w_p \in D_t$  are such that the representing map takes  $w_i$  to  $a_i$ . ■

There is also a representation theorem for metric algebras, that is topological algebras which are metric spaces, which we state without proof.

**Theorem 4.5.17.** *Each metric  $\Sigma$ -algebra is domain representable.*

The theorem is proved with techniques similar to the ones above, by considering 'converging' ideals.

## 5 Effective domains

In this section we discuss the theory of effectively approximable structures. The class of Scott–Ershov domains is of primary importance for our approach and is also of independent interest for computer science. Thus we go through the basic theory of effective domains and constructive subdomains in Sections 5.1 and 5.2. Then, in Section 5.3, we apply the theory to effectively approximable algebras using domain representability. In particular, we establish the precise connections with notions from recursive analysis.

In Sections 5.4 and 5.5 we return to the theory of effective domains and prove generalisations of the Myhill–Shepherdson Theorem and the Kreisel–Lacombe–Shoenfield Theorem. As an easy consequence we obtain a version of Ceitin’s Theorem for certain effectively approximable topological spaces, such as the reals  $\mathbb{R}$ .

A prerequisite for the understanding of this section is a knowledge of basic recursion theory. Our notation is standard. Thus  $W_e$  denotes the recursively enumerable or r.e. set of natural numbers with index  $e$ . We denote the set of natural numbers by  $\omega$  or by  $\mathbb{N}$ .

## 5.1 Basic theory

The first question to settle is: *what should we mean by an effective domain?* More generally, what should we mean by a structure together with an approximation being effective? Consider a set  $A$  together with an approximation  $P = (P; \sqsubseteq)$  for  $A$ . Then to say that we have an effective approximation for an element  $x$  in  $A$  is to say that there is a computable sequence of approximations from  $P$  converging to  $x$ . In order to speak of computable sequences of approximations we require, at least, that the approximation structure  $P$  be computable in the sense of Section 1.4. The approximation structure for a domain  $D$  is the  $\text{cul } D_c$  of compact elements. Thus what we require of an effective domain is that the  $\text{cul}$  of compact elements be a computable structure. Here is the precise definition.

**Definition 5.1.1.** Let  $D$  be a domain. Then  $(D, \alpha)$  is an *effective domain* if  $(D_c, \alpha)$  is a computable  $\text{cul}$ , where  $D_c$  is the structure

$$D_c = (D_c; \sqsubseteq, \text{Cons}, \sqcup, \perp)$$

of compact elements of  $D$ , and  $\text{Cons}$  is the binary consistency relation on  $D_c$ .

We say that a domain  $D$  is *effective* if  $(D, \alpha)$  is an effective domain for some  $\alpha$ .

Clearly any countable set  $A$  gives rise to a computable  $\text{cul } A_\perp$ , the flat domain over  $A$ , which also is an effective domain. More interesting examples are the domains  $\mathcal{P}$  and  $\wp(\omega)$  of partial functions on and subsets of  $\omega$ , respectively, ordered by inclusion. The compact elements for these domains are the finite functions and the finite sets, respectively. These can be numbered in an obvious way making the domains effective. Also the domains representing the inverse limit construction on the examples in 4.3.2 are effective, as is the standard domain representation of the reals  $\mathbb{R}$ .

By a *presentation* of the effective domain  $(D, \alpha)$  we mean a presentation of the computable  $\text{cul } (D_c, \alpha)$ . Thus a presentation of  $(D, \alpha)$  is a structure

$$\Omega_\alpha = (\Omega_\alpha; \hat{\sqsubseteq}, \widehat{\text{Cons}}, \hat{\sqcup}, \hat{\perp})$$

where  $\Omega_\alpha$  is a recursive set of natural numbers, the relations  $\hat{\sqsubseteq}$  and  $\widehat{\text{Cons}}$  are recursive, the operation  $\hat{\sqcup}$  is recursive and  $\hat{\perp}$  is a distinguished number (a constant), and  $\alpha : \Omega_\alpha \rightarrow D_c$  is a surjective homomorphism.

Let  $(P, \alpha)$  be a computable csl and let  $\bar{P}$  be the ideal completion of  $P$ . Define  $\hat{\alpha} : \Omega_\alpha \rightarrow \bar{P}_c$  by  $\hat{\alpha}(n) = [\alpha(n)]$ , the principal ideal generated by  $\alpha(n)$ . From the proof of the Representation Theorem 4.2.5 it follows that  $\hat{\alpha}$  is a computable numbering of  $\bar{P}_c$  and, in fact, that a presentation of  $(P, \alpha)$  is also a presentation of  $(\bar{P}_c, \hat{\alpha})$  and hence of the effective domain  $(\bar{P}, \hat{\alpha})$ . We make this observation explicit.

**Proposition 5.1.2.** *If  $(P, \alpha)$  is a computable csl then  $(\bar{P}, \hat{\alpha})$  is an effective domain. Furthermore, each presentation of  $(P, \alpha)$  is also a presentation of  $(\bar{P}, \hat{\alpha})$ .*

We usually identify  $\alpha$  with  $\hat{\alpha}$  and  $P$  with the principal ideals in  $\bar{P}$ .

Let  $(D, \alpha)$  be an effective domain. We know that each element of  $D$  is obtained as the limit of a sequence of compact elements. Of course, such a sequence need not be computable. In fact, since there are only countably many computable sequences, most elements in an uncountable domain do not have effective approximations. Let  $x \in D$  and consider an  $\alpha$ -computable sequence  $f : \omega \rightarrow D_c$  such that  $x = \bigsqcup \{f(n) : n \in \omega\}$ , and let  $\hat{f} : \omega \rightarrow \Omega_\alpha$  be a recursive tracking function of  $f$ . We have

$$a \in \text{approx}(x) \Leftrightarrow \exists n (a \sqsubseteq f(n))$$

and hence

$$m \in \alpha^{-1}[\text{approx}(x)] \Leftrightarrow \exists n (m \hat{\sqsubseteq} \hat{f}(n))$$

which shows that  $\alpha^{-1}[\text{approx}(x)]$  is r.e., that is  $\text{approx}(x)$  is  $\alpha$ -semidecidable. Conversely, if  $\text{approx}(x)$  is  $\alpha$ -semidecidable let  $\hat{f} : \omega \rightarrow \Omega_\alpha$  be a total recursive function enumerating  $\alpha^{-1}[\text{approx}(x)]$ . Then the function  $f : \omega \rightarrow D_c$  tracked by  $\hat{f}$  is a computable sequence whose limit is  $x$ . The uniformities implicit in the above constructions motivate the following definition.

**Definition 5.1.3.** Let  $(D, \alpha)$  be an effective domain. Then  $x \in D$  is an  $\alpha$ -computable element if  $\alpha^{-1}[\text{approx}(x)]$  is r.e. An r.e. index of  $\alpha^{-1}[\text{approx}(x)]$  is an  $\alpha$ -index of the computable element  $x$ .

Other terms for computable in the literature are *recursive* or *effective*. Note that each compact element in an effective domain is computable.

A continuous function between domains should be effective if its values on computable elements can be effectively approximated. To capture this, we recall that a continuous function  $f : D \rightarrow E$  is determined by its actions on the compact elements in the following sense. For  $x \in D$  consider  $a \in \text{approx}(x)$  and ‘compute’ each  $b \in \text{approx}(f(a))$ . The value  $f(x)$  is then the limit of all possible such  $b$ ’s. That is,



$$f(x) = \bigsqcup \{b \in E_c : (\exists a \in \text{approx}(x))(b \sqsubseteq f(a))\}.$$

Thus  $f$  is completely described by the relation  $R_f \subseteq D_c \times E_c$  defined by

$$R_f(a, b) \Leftrightarrow b \sqsubseteq f(a).$$

This motivates the following definition.

**Definition 5.1.4.** Let  $(D, \alpha)$  and  $(E, \beta)$  be effective domains. A continuous function  $f : D \rightarrow E$  is  $(\alpha, \beta)$ -effective if the relation  $R_f \subseteq D_c \times E_c$  is  $(\alpha, \beta)$ -semidecidable, that is the relation

$$\hat{R}_f(m, n) \Leftrightarrow R_f(\alpha(m), \beta(n))$$

is r.e. An r.e. index for  $\hat{R}_f$  is an *effective index* for  $f$  with respect to  $\alpha$  and  $\beta$ .

**Proposition 5.1.5.** Let  $(D, \alpha)$ ,  $(E, \beta)$ , and  $(F, \gamma)$  be effective domains and let  $f : D \rightarrow E$  and  $g : E \rightarrow F$  be continuous, and  $(\alpha, \beta)$ -effective and  $(\beta, \gamma)$ -effective respectively.

1. If  $x \in D$  is  $\alpha$ -computable then  $f(x) \in E$  is  $\beta$ -computable.
2. The composition  $h = g \circ f : D \rightarrow F$  is  $(\alpha, \gamma)$ -effective.

**Proof.** 1. Let  $\hat{R}$  be an r.e. tracking relation for  $f$ . Suppose  $x \in D$  is  $\alpha$ -computable. Then  $V = \alpha^{-1}[\text{approx}(x)]$  is r.e. and hence

$$W = \{n \in \omega : (\exists m \in V) \hat{R}(m, n)\}$$

is r.e. We claim that  $W = \beta^{-1}[\text{approx}(f(x))]$ . Let  $n \in W$  and choose  $m \in V$  such that  $\hat{R}(m, n)$ , that is  $\beta(n) \sqsubseteq f(\alpha(m))$ . But  $f(\alpha(m)) \sqsubseteq f(x)$  by the monotonicity of  $f$  so  $\beta(n) \in \text{approx}(f(x))$ . For the converse inclusion assume  $\beta(n) \in \text{approx}(f(x))$ . Then

$$\beta(n) \sqsubseteq f(x) = \bigsqcup \{f(a) : a \in \text{approx}(x)\}$$

by the continuity of  $f$  and hence there is an  $a \in \text{approx}(x)$  such that  $\beta(n) \sqsubseteq f(a)$ . But then there is  $m \in V$  such that  $\hat{R}(m, n)$ , so  $n \in W$ .

2. Suppressing the numberings and letting  $a, b$ , and  $d$  vary over  $D_c$ ,  $E_c$ , and  $F_c$  respectively we have by the continuity of  $g$ ,

$$d \sqsubseteq g(f(a)) \Leftrightarrow (\exists b \sqsubseteq f(a))(d \sqsubseteq g(b)).$$

The right hand side is semidecidable by the effectivity of  $f$  and  $g$ . ■

We observe that the above proof is *uniform*. Thus an r.e.-index of  $W$  is obtained uniformly from r.e.-indices of  $V$  and  $\hat{R}$ , i.e. an index of  $f(x)$

is obtained uniformly from indices of  $x$  and  $f$ . Similarly an index of  $h$  is obtained uniformly from indices of  $f$  and  $g$ .

All the usual domain constructions are finitary. It therefore easily follows that effectivity of domains is preserved under these constructions.

**Theorem 5.1.6.** *Let  $(D, \alpha)$  and  $(E, \beta)$  be effective domains. Then  $D \times E$ ,  $[D \rightarrow E]$ ,  $D + E$ ,  $D \otimes E$ ,  $D \oplus E$ ,  $[D \rightarrow_{\perp} E]$ , and  $D_{\perp}$  are effective domains with presentations obtained uniformly from presentations of  $(D, \alpha)$  and  $(E, \beta)$ .*

To minimise notation we will always assume that the numbering of the effective domain  $D \times E$  we consider is the one obtained from the given effective domains  $(D, \alpha)$  and  $(E, \beta)$  as in the above theorem, and similarly for the other domain constructions.

It is also the case that any domain equation, formed using the operations above and having effective parameters, has an effective solution. The reason is, again, that these operations are finitary. We note this important well-known fact as a theorem, while referring to Smyth and Plotkin [1982] or Stoltenberg-Hansen *et al.* [1994] for a proof.

**Theorem 5.1.7.** *The least solution of a domain equation built up by  $\rightarrow$ ,  $\times$ ,  $+$ ,  $\otimes$ ,  $\oplus$ ,  $\rightarrow_{\perp}$ , and  $\perp$  and using effective domains as parameters is an effective domain. A presentation of this solution is obtained effectively from the given equation.*

Recall that the compact elements in the domain  $[D \rightarrow E]$  are precisely the suprema  $\bigsqcup_{i < n} \langle a_i; b_i \rangle$  of finitely many consistent step functions  $\langle a_i; b_i \rangle$ , with  $a_i \in D_c$  and  $b_i \in E_c$ , where

$$\langle a; b \rangle(x) = \begin{cases} b & \text{if } a \sqsubseteq x \\ \perp & \text{otherwise.} \end{cases}$$

The following is an important conceptual observation.

**Proposition 5.1.8.** *Let  $(D, \alpha)$  and  $(E, \beta)$  be effective domains. Then a continuous function  $f : D \rightarrow E$  is effective if and only if  $f$  is a computable element in  $[D \rightarrow E]$ .*

**Proof.** We suppress the numberings in the argument. For an arbitrary compact element  $\bigsqcup_{i < n} \langle a_i; b_i \rangle$  in  $[D \rightarrow E]$  we have

$$\bigsqcup_{i < n} \langle a_i; b_i \rangle \sqsubseteq f \Leftrightarrow (\forall i < n)(\langle a_i; b_i \rangle \sqsubseteq f) \Leftrightarrow (\forall i < n)(b_i \sqsubseteq f(a_i))$$

and the latter is semidecidable whenever  $f$  is effective. Thus  $f$  is a computable element in  $[D \rightarrow E]$  when  $f$  is effective. For the converse we need only note that for  $a \in D_c$ ,  $b \in E_c$ ,

$$b \sqsubseteq f(a) \Leftrightarrow \langle a; b \rangle \in \text{approx}(f).$$

All the continuous functions associated with our domain constructions, such as *eval*, *curry*, the projection functions, and so on, are effective. We verify this for *eval*, leaving the remaining ones as easy exercises. Suppressing numberings we assume that  $D$  and  $E$  are effective domains and consider  $eval : [D \rightarrow E] \times D \rightarrow E$ , defined by  $eval(f, x) = f(x)$ . We have

$$\begin{aligned} b \sqsubseteq eval(\bigsqcup_{i < n} \langle a_i; b_i \rangle, a) &\Leftrightarrow b \sqsubseteq (\bigsqcup_{i < n} \langle a_i; b_i \rangle)(a) \\ &\Leftrightarrow b \sqsubseteq \bigsqcup \{b_i : a_i \sqsubseteq a\} \end{aligned}$$

which is even decidable.

Recall that each continuous function  $f : D \rightarrow D$  has a least fixed point. Let  $fix : [D \rightarrow D] \rightarrow D$  be the fixed point operator defined by

$$fix(f) = \text{least fixed point of } f.$$

**Proposition 5.1.9.** *Let  $(D, \alpha)$  be an effective domain. Then  $fix : [D \rightarrow D] \rightarrow D$  is continuous and effective.*

**Proof.** Define  $h_n : [D \rightarrow D] \rightarrow D$  for  $n \in \omega$  by

$$\begin{aligned} h_0(f) &= \perp \\ h_{n+1}(f) &= eval(f, h_n(f)). \end{aligned}$$

It is easy to see that each  $h_n$  is continuous and that  $fix = \bigsqcup \{h_n : n \in \omega\}$ . It follows that  $fix$  is continuous. For effectivity, it suffices to show that each  $h_n$  is effective uniformly in  $n$ . But this follows immediately by the remarks above. ■

**Corollary 5.1.10.** *Let  $(D, \alpha)$  be an effective domain. If  $f : D \rightarrow D$  is effective then  $fix(f)$  is a computable element in  $D$ .*

We extend the notion of effectivity to structured domains in the obvious way.

**Definition 5.1.11.** A  $\Sigma$ -domain  $D = (D; \sqsubseteq, \perp; x_1, \dots, x_p, \Psi_1, \dots, \Psi_q)$  is *effective* if  $(D; \sqsubseteq, \perp)$  is an effective domain, each  $x_i$  is a computable element, and each operation  $\Psi_i : D^{n_i} \rightarrow D$  is effective.

Let  $A$  be a computable  $\Sigma$ -algebra. Then the  $\Sigma$ -domain  $A_\perp$  is effective, where the operations in  $A_\perp$  are the strict extensions of the operations in  $A$ . More interestingly, suppose  $A$  is a computable  $\Sigma$ -algebra and suppose  $\{\equiv_n\}_{n \in \omega}$  is a decidable family of separating congruences, uniformly in  $n$ . Then the  $\Sigma$ -domain  $D(A)$  representing  $A$  is effective. Similarly the standard structured domain representing the ring of real numbers  $\mathbb{R}$  is effective.

Finally, we note that the class of effective  $\Sigma$ -domains is closed under the usual constructions, just as in Theorem 5.1.6.

## 5.2 Constructive subdomains

In this section we study subsets of an effective domain containing only computable elements and having a numbering amenable to the numbering of the domain. In particular, we study the subset of all computable elements, which has nice effective closure properties.

**Definition 5.2.1.** Let  $(D, \alpha)$  be an effective domain and suppose  $D_c \subseteq B \subseteq D$ . Then  $(B, \gamma)$  is a *constructive subdomain* of  $(D, \alpha)$  if  $\gamma : \Omega_\gamma \rightarrow B$  is a surjective numbering such that  $\Omega_\gamma \subseteq \omega$  is recursive, and

1. the inclusion mapping  $\iota : D_c \rightarrow B$  is  $(\alpha, \gamma)$ -computable, and
2. the relation  $R(n, m) \Leftrightarrow \alpha(n) \sqsubseteq \gamma(m)$  is r.e., that is  $\text{approx}(\gamma(m))$  is  $\alpha$ -semidecidable uniformly in  $m$ .

We say that  $\gamma$  is a *constructive numbering* of  $B$  if  $(B, \gamma)$  is a constructive subdomain. Trivially,  $(D_c, \alpha)$  is a constructive subdomain of  $(D, \alpha)$ .

In the literature one often considers constructive domains  $(B, \gamma, \alpha)$  outright without reference to containment in a domain  $D$ . These are also called *constructive  $f_0$ -spaces*, following Ershov. However, using an ideal completion one can easily construct an effective domain  $(D, \alpha)$  such that  $(B, \gamma)$  is a constructive subdomain of  $(D, \alpha)$ . Thus our situation is completely general.

The  $\gamma$ -equality relation is not decidable in general. The following observation is the best possible.

**Proposition 5.2.2.** Let  $(B, \gamma)$  be a constructive subdomain of  $(D, \alpha)$ . Then the equality relation  $\equiv_\gamma$  is a  $\prod_2^0$ -relation.

**Proof.** We have

$$\begin{aligned} n \equiv_\gamma m &\Leftrightarrow \gamma(n) = \gamma(m) \\ &\Leftrightarrow \text{approx}(\gamma(n)) = \text{approx}(\gamma(m)) \\ &\Leftrightarrow \alpha^{-1}[\text{approx}(\gamma(n))] = \alpha^{-1}[\text{approx}(\gamma(m))] \\ &\Leftrightarrow \forall k (\alpha(k) \sqsubseteq \gamma(n) \Leftrightarrow \alpha(k) \sqsubseteq \gamma(m)) \end{aligned}$$

which is easily seen to be  $\prod_2^0$  using Definition 5.2.1 (2). ■

The most interesting constructive subdomain is the one consisting of all computable elements.

**Definition 5.2.3.** Let  $(D, \alpha)$  be an effective domain. Then

$$D_k = \{x \in D : x \text{ is } \alpha\text{-computable}\}.$$

**Theorem 5.2.4.** Let  $(D, \alpha)$  be an effective domain. Then there is a numbering  $\bar{\alpha} : \omega \rightarrow D_k$  such that  $(D_k, \bar{\alpha})$  is a constructive subdomain of  $(D, \alpha)$ .

**Proof.** Consider some fixed standard enumeration  $\{W_e : e \in \omega\}$  of the r.e. sets where  $W_e^n$  denotes (a canonical index for) the finite part of  $W_e$

generated by stage  $n$  in the standard enumeration. Define a function  $f : \wp_f(\omega) \rightarrow \omega$  where  $\wp_f(\omega)$  denotes the set of finite subsets of  $\omega$ , by

$$f(K) = \begin{cases} \mu n[\alpha(n) = \bigsqcup \alpha[K]] & \text{if } \alpha[K] \text{ consistent} \\ 0 & \text{otherwise.} \end{cases}$$

Then  $f$  is recursive since  $(D, \alpha)$  is an effective domain. Next we effectively define finite sets  $V_e^n$  by

$$V_e^0 = \{n_0\} \text{ where } \alpha(n_0) = \perp, \text{ and}$$

$$V_e^{n+1} = \begin{cases} V_e^n \cup W_e^n \cup \{f(V_e^n \cup W_e^n)\} & \text{if } \alpha[V_e^n \cup W_e^n] \text{ consistent} \\ V_e^n & \text{otherwise.} \end{cases}$$

Let  $V_e = \bigcup \{V_e^n : n \in \omega\}$ . Then  $V_e$  is an r.e. set since a canonical index of  $V_e^n$  is obtained recursively from  $e$  and  $n$ . Note that  $\alpha[V_e^n]$  is a directed set for each  $n$  and hence  $\alpha[V_e]$  is directed. Thus we may define  $\bar{\alpha}(e) = \bigsqcup \alpha[V_e]$ .

Suppose  $x \in D_k$  and let  $W_e = \alpha^{-1}[\text{approx}(x)]$ . Clearly  $V_e^0 \subseteq W_e$  since  $\perp \in \text{approx}(x)$ . Assume inductively that  $V_e^n \subseteq W_e$ . Then  $\alpha[V_e^n \cup W_e^n]$  is consistent since it is bounded by  $x$ . But then  $\alpha(f(V_e^n \cup W_e^n)) \in \text{approx}(x)$  and hence  $V_e^{n+1} \subseteq W_e$ . Thus  $V_e \subseteq W_e$ . But then  $V_e^n \cup W_e^n \subseteq W_e$  for each  $n$  and hence  $\alpha[V_e^n \cup W_e^n]$  is consistent, again since it is bounded by  $x$ . It follows that  $V_e = W_e$ . In particular,  $\bar{\alpha}(e) = x$ .

On the other hand, for  $e \in \omega$  we have

$$\alpha(n) \in \text{approx}(\bar{\alpha}(e)) \Leftrightarrow (\exists m \in V_e)(\alpha(n) \sqsubseteq \alpha(m)) \quad (*)$$

since  $\text{approx}(\bigsqcup A) = \bigcup \{\text{approx}(y) : y \in A\}$  for each directed set  $A$ . Thus  $\bar{\alpha}(e) \in D_k$ . We have shown that  $\bar{\alpha} : \omega \rightarrow D_k$  is a surjective numbering.

To show that  $(D_k, \bar{\alpha})$  is a constructive subdomain consider first condition (2) of Definition 5.2.1. We have

$$\alpha(n) \sqsubseteq \bar{\alpha}(e) \Leftrightarrow \alpha(n) \in \text{approx}(\bar{\alpha}(e))$$

since  $\alpha(n)$  is compact, and this is an r.e. relation by  $(*)$  and the uniform construction of  $V_e$ . Finally to see that the inclusion  $\iota : D_c \rightarrow D_k$  is  $(\alpha, \bar{\alpha})$ -computable it suffices to note that the set  $\{n : \alpha(n) \sqsubseteq \alpha(m)\}$  is r.e. with an index  $e$  obtained uniformly from  $m$ . That is, there is a total recursive function  $\hat{\iota}$  such that  $W_{\hat{\iota}(m)} = \{n : \alpha(n) \sqsubseteq \alpha(m)\}$ . But then we have  $V_{\hat{\iota}(m)} = W_{\hat{\iota}(m)}$  and  $\alpha(m) = \bigsqcup \alpha[V_{\hat{\iota}(m)}] = \bar{\alpha}(\hat{\iota}(m))$  so that  $\iota : D_c \rightarrow D_k$  is tracked by  $\hat{\iota}$ . ■

We refer to  $\bar{\alpha}$ , the numbering obtained in Theorem 5.2.4, as the *canonical numbering* of  $D_k$ . We shall see, in Corollary 5.2.12, that all recursively complete numberings of  $D_k$  are recursively equivalent to  $\bar{\alpha}$ . Thus, when studying  $D_k$ , we may without loss of generality work with  $\bar{\alpha}$ .



Here we isolate some simple but useful properties of  $\bar{\alpha}$  from the proof of Theorem 5.2.4.

**Lemma 5.2.5.** *Let  $(D, \alpha)$  be an effective domain and let  $\bar{\alpha}$  be the canonical numbering of  $D_k$ .*

1. *There is a total recursive function  $t : \omega \rightarrow \omega$  such that for each  $e \in \omega$ ,  $W_{t(e)} = \alpha^{-1}[\text{approx}(\bar{\alpha}(e))]$  and  $\bar{\alpha}(t(e)) = \bar{\alpha}(e)$ .*
2.  *$W_e = \alpha^{-1}[\text{approx}(x)] \Rightarrow \bar{\alpha}(e) = x$ .*
3. *If  $\alpha[W_e]$  is directed then  $\bar{\alpha}(e) = \bigcup \alpha[W_e]$ .*

The following lemma describes a useful way effectively to approximate  $\bar{\alpha}(e)$ .

**Lemma 5.2.6.** *Let  $(D, \alpha)$  be an effective domain. Then there is a function  $\tilde{\alpha} : \omega^2 \rightarrow D_c$  and a total recursive function  $s : \omega^2 \rightarrow \omega$  such that for each  $e \in \omega$ :*

1.  *$m \leq n \Rightarrow \tilde{\alpha}(e, m) \subseteq \tilde{\alpha}(e, n)$ ,*
2.  *$\bar{\alpha}(e) = \bigcup_{n \in \omega} \tilde{\alpha}(e, n)$ , and*
3.  *$\tilde{\alpha}(e, n) = \alpha(s(e, n))$ .*

**Proof.** Using the notation in the proof of Theorem 5.2.4 we set

$$\tilde{\alpha}(e, n) = \bigcup \alpha[V_e^n] \text{ and } s(e, n) = \mu k (\alpha(k) \in \bigcup \alpha[V_e^n]).$$

■

**Proposition 5.2.7.** *Suppose  $(B, \gamma)$  is a constructive subdomain of  $(D, \alpha)$ . Then  $B \subseteq D_k$  and the inclusion  $\iota : B \rightarrow D_k$  is  $(\gamma, \bar{\alpha})$ -computable.*

**Proof.** For each  $e \in \Omega_\gamma$  the set  $W = \{n \in \omega : \alpha(n) \subseteq \gamma(e)\}$  is r.e. uniformly in  $e$ , since  $\gamma$  is a constructive numbering. Thus there is a total recursive function  $h$  such that  $\alpha^{-1}[\text{approx}(\gamma(e))] = W_{h(e)}$ . But then, by Lemma 5.2.5,  $\bar{\alpha}(h(e)) = \gamma(e)$ . Thus  $\gamma(e) \in D_k$  and  $h$  tracks the inclusion  $\iota : B \rightarrow D_k$ . ■

The constructive subdomain  $D_k$  of  $(D, \alpha)$  is of central interest in our study. It is of course in general not a domain since it may not be complete. However,  $D_k$  is the unique constructive subdomain which is recursively complete. Let  $B$  be a set with a numbering  $\gamma$ . Then a set  $A \subseteq B$  is said to be *weakly  $\gamma$ -semidecidable* if  $A = \gamma[W_e]$  for some  $e$ .

**Definition 5.2.8.** A constructive subdomain  $(B, \gamma)$  of an effective domain  $(D, \alpha)$  is *recursively complete* if  $\bigcup A \in B$  whenever  $A \subseteq B$  is a weakly  $\gamma$ -semidecidable directed set.

**Proposition 5.2.9.** *Let  $(D, \alpha)$  be an effective domain.*

1. *If the constructive subdomain  $(B, \gamma)$  is recursively complete then  $B = D_k$ .*

2. If  $\gamma$  is a constructive numbering of  $D_k$  then  $(D_k, \gamma)$  is recursively complete.

**Proof.** To prove (1) let  $x \in D_k$ . Then  $\alpha^{-1}[\text{approx}(x)]$  is an r.e. set and hence, since the inclusion  $\iota : D_c \rightarrow B$  is  $(\alpha, \gamma)$ -computable,  $\text{approx}(x)$  is weakly  $\gamma$ -semidecidable. But then  $x = \bigsqcup \text{approx}(x) \in B$ , since  $B$  is recursively complete.

To prove (2) assume that  $\gamma$  is a constructive numbering of  $D_k$ . As previously remarked,  $\text{approx}(\bigsqcup A) = \bigcup \{\text{approx}(x) : x \in A\}$  for directed sets  $A$ . Suppose  $A = \gamma[W]$  is a directed set, where  $W$  is an r.e. set. Then

$$\alpha^{-1}[\text{approx}(\bigsqcup A)] = \{n \in \omega : (\exists m \in W)(\alpha(n) \sqsubseteq \gamma(m))\}$$

and hence  $\alpha^{-1}[\text{approx}(\bigsqcup A)]$  is r.e. In particular,  $\bigsqcup A \in D_k$ . ■

**Definition 5.2.10.** Let  $\gamma$  be a constructive numbering of  $D_k$ . Then  $\gamma$  is a *recursively complete numbering* if there is a total recursive function  $h$  such that if  $\gamma[W_e]$  is a directed set then  $\gamma(h(e)) = \bigsqcup \gamma[W_e]$ .

**Theorem 5.2.11.** Let  $(D, \alpha)$  be an effective domain and let  $\bar{\alpha}$  be the canonical numbering of  $D_k$ . Then  $\bar{\alpha}$  is a recursively complete numbering. Furthermore, if  $\gamma$  is a recursively complete numbering of  $D_k$  then the identity  $\text{id} : D_k \rightarrow D_k$  is  $(\bar{\alpha}, \gamma)$ -computable.

**Proof.** Let  $h$  be a total recursive function such that

$$W_{h(e)} = \{n \in \omega : (\exists m \in W_e)(\alpha(n) \sqsubseteq \bar{\alpha}(m))\}.$$

Then  $W_{h(e)} = \alpha^{-1}[\text{approx}(\bigsqcup \bar{\alpha}[W_e])]$  whenever  $\bar{\alpha}[W_e]$  is a directed set, and  $\bar{\alpha}(h(e)) = \bigsqcup \bar{\alpha}[W_e]$  by Lemma 5.2.5 (2).

Now suppose that  $\gamma$  is a recursively complete numbering of  $D_k$  and  $h$  is the associated total recursive function. Since  $\gamma$  is a constructive numbering there is a recursive function  $g$  such that  $\gamma[W_{g(e)}] = \text{approx}(\bar{\alpha}(e))$  for each  $e$ . But then  $\bar{\alpha}(e) = \bigsqcup \gamma[W_{g(e)}] = \gamma(h(g(e)))$  so that  $h \circ g$  tracks the identity. ■

**Corollary 5.2.12.**  $D_k$  is stable with respect to recursively complete numberings in the sense that all recursively complete numberings are recursively equivalent.

**Proof.** Given recursively complete numberings  $\beta$  and  $\gamma$  of  $D_k$  we must show that there exists a recursive function  $h : \Omega_\beta \rightarrow \Omega_\gamma$  such that for each  $n \in \Omega_\beta$ ,  $\beta(n) = \gamma(h(n))$ . But this follows from Proposition 5.2.7 and Theorem 5.2.11. ■

The results above state that up to recursive equivalence there is one and only one recursively complete subdomain of an effective domain as long as we require the numberings to be recursively complete. Therefore, given

an effective domain  $(D, \alpha)$ , we will henceforth without loss of generality consider the recursively complete subdomain  $(D_k, \bar{\alpha})$ .

As a final observation we note that continuous effective functions, when restricted to the constructive or computable elements, are computable with respect to the canonical numberings.

**Proposition 5.2.13.** *Let  $(D, \alpha)$  and  $(E, \beta)$  be effective domains. If  $f : D \rightarrow E$  is a continuous  $(\alpha, \beta)$ -effective function then  $f|_{D_k} : D_k \rightarrow E_k$  is  $(\bar{\alpha}, \bar{\beta})$ -computable.*

**Proof.** That  $f|_{D_k}$  takes its values in  $E_k$  is the content of Proposition 5.1.5 (1). Using the approximations of Lemma 5.2.6 and the continuity of  $f$  we have

$$f(\bar{\alpha}(e)) = f\left(\bigsqcup_{n \in \omega} \tilde{\alpha}(e, n)\right) = \bigsqcup_{n \in \omega} f(\tilde{\alpha}(e, n))$$

and hence

$$\beta^{-1}[\text{approx}(f(\bar{\alpha}(e)))] = \{m \in \omega : \exists n(\beta(m) \sqsubseteq f(\alpha(s(e, n))))\}.$$

It follows from the effectivity of  $f$  that there is a total recursive function  $h$  such that  $W_{h(e)} = \beta^{-1}[\text{approx}(f(\bar{\alpha}(e)))]$ . But then  $\bar{\beta}h(e) = f(\bar{\alpha}(e))$  by Lemma 5.2.5, so that  $h$  tracks  $f|_{D_k}$  with respect to  $(\bar{\alpha}, \bar{\beta})$ . ■

The converse of Proposition 5.2.13 is the content of a generalised version of the Myhill-Shepherdson Theorem.

### 5.3 Algebras effectively approximable by domains

The general method we pursue to study the effective properties of a generally uncountable topological algebra  $A$  is to find an *effective* domain  $D$  representing  $A$  in the sense of Definition 4.2.10 and then measure the effectivity of  $A$  by means of the effectivity of the representing domain  $D$ . Thus the effectivity of  $A$  is dependent on the domain representation  $D$  and its effectivity, just as the computability of a countable algebra is dependent on the particular numbering considered. In practice, given an algebra  $A$  one finds a computable structure  $P$  of ‘concrete’ approximations for  $A$  which is such that the ideal completion  $\bar{P}$  of  $P$  is a domain representation of  $A$ .

**Definition 5.3.1.** Let  $A$  be a topological  $\Sigma$ -algebra.

1.  $A$  is *effectively domain representable* by  $(D, D_A, v, \alpha)$  if  $(D, D_A, v)$  is a domain representation of  $A$  and  $(D, \alpha)$  is an effective  $\Sigma$ -domain.
2.  $A$  is *effectively domain representable* if  $A$  is representable by an effective domain.

The effective versions of Propositions 4.2.11 and 4.2.12 hold.

**Proposition 5.3.2.** *Let  $A$  and  $B$  be topological  $\Sigma$ -algebras which are effectively domain representable. Then*

1.  $A \times B$  is effectively domain representable;
2. each subalgebra  $C$  of  $A$  is effectively domain representable; and
3. each continuous homomorphic image  $C$  of  $A$  is effectively domain representable.

**Proof.** (1) follows from Proposition 4.2.11 and Theorem 5.1.6. (2) and (3) follow directly from Proposition 4.2.12. ■

Next we consider the set of computable elements of an effectively domain representable algebra  $A$ , analogous to the set  $D_k$  of computable elements in an effective domain  $D$  (Definition 5.2.3).

**Definition 5.3.3.** Let  $A$  be a topological  $\Sigma$ -algebra effectively domain representable by  $(D, D_A, v, \alpha)$ . Then the set  $A_k$  of  $(D, D_A, v, \alpha)$ -computable elements of  $A$  is the set

$$A_k = \{x \in A : v^{-1}(x) \cap D_k \neq \emptyset\}.$$

As usual we suppress the reference to  $(D, D_A, v, \alpha)$  whenever possible and simply write  $A_k$  for the set of computable elements, the effective domain representation being understood.

A  $\Sigma$ -algebra  $A$  is said to have a *numbering with recursive operations* if there is a surjection  $\beta : \Omega \rightarrow A$ , where  $\Omega \subseteq \omega$ , such that each operation in  $A$  is  $\beta$ -computable. By a  $k$ -ary operation  $\sigma$  on  $A$  being  $\beta$ -computable we mean that there is a *partial* recursive function  $\hat{\sigma}$  such that

$$n_1, \dots, n_k \in \Omega \Rightarrow \hat{\sigma}(n_1, \dots, n_k) \text{ defined,}$$

and for  $n_1, \dots, n_k \in \Omega$ ,

$$\sigma(\beta(n_1), \dots, \beta(n_k)) = \beta(\hat{\sigma}(n_1, \dots, n_k)),$$

that is  $\hat{\sigma}$  tracks  $\sigma$  with respect to  $\beta$  in the usual way.

We say that  $(A, \beta)$  is a *numbered algebra with recursive operations* if  $\beta$  is a numbering of  $A$  with recursive operations. Note that we put *no* requirement on the complexity of the code set  $\Omega$  nor on the (relative) complexity of the equality relation.

**Proposition 5.3.4.** *Let  $A = (A; a_1, \dots, a_p, \sigma_1, \dots, \sigma_q)$  be an effectively domain representable topological  $\Sigma$ -algebra.*

1.  $A_k$  is a subalgebra of  $A$ .
2.  $A_k$  is a numbered algebra with recursive operations.

**Proof.** Let  $(D, D_A, v, \alpha)$  be an effective domain representation of  $A$ . Recall from Theorem 5.2.4 the canonical numbering  $\bar{\alpha} : \omega \rightarrow D_k$  making

$(D_k, \bar{\alpha})$  into a constructive subdomain of  $(D, \alpha)$ . Let  $\Omega_A = \bar{\alpha}^{-1}[D_k \cap D_A]$  and define  $\tilde{\alpha} : \Omega_A \rightarrow A_k$  by

$$\tilde{\alpha}(n) = v(\bar{\alpha}(n)).$$

Then  $\tilde{\alpha}$  is a surjective numbering of  $A_k$ .

Clearly each  $a_i$  has an  $\tilde{\alpha}$ -index. Thus it remains to show that for each operation  $\sigma$  in  $A$ , the restriction of  $\sigma$  to  $A_k$  is an  $\tilde{\alpha}$ -computable operation. Say that  $\sigma$  is an  $m$ -ary such operation and let  $\hat{\sigma}$  be the continuous effective operation in  $D$  representing  $\sigma$ .

First we show that  $A_k$  is closed under  $\sigma$ . Let  $b_1, \dots, b_m \in A_k$  and choose  $x_1, \dots, x_m \in D_k \cap D_A$  such that  $v(x_i) = b_i$ . Then, by Proposition 5.1.5,  $\hat{\sigma}(x_1, \dots, x_m) \in D_k \cap D_A$  and hence

$$\sigma(b_1, \dots, b_m) = v(\hat{\sigma}(x_1, \dots, x_m)) \in A_k,$$

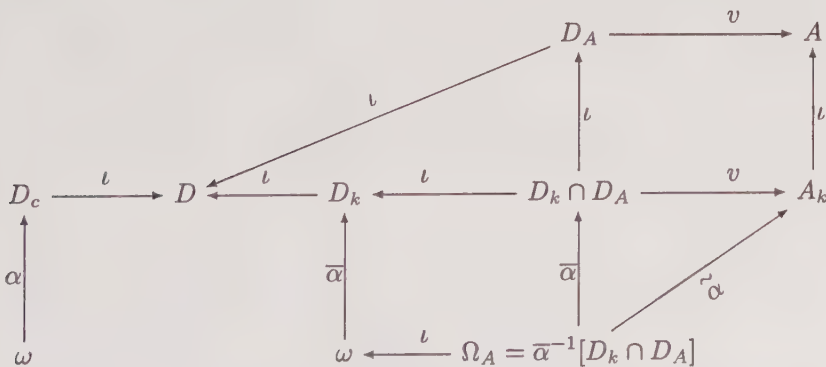
proving that  $A_k$  is a subalgebra of  $A$ .

By Proposition 5.2.13, the restriction of  $\hat{\sigma}$  to  $D_k$  is  $\bar{\alpha}$ -computable, i.e. there is a recursive function  $f : \omega^m \rightarrow \omega$  tracking  $\hat{\sigma}$ . Thus for each  $n_1, \dots, n_m \in \Omega_A$ ,

$$\begin{aligned} \sigma(\tilde{\alpha}(n_1), \dots, \tilde{\alpha}(n_m)) &= v(\hat{\sigma}(\bar{\alpha}(n_1), \dots, \bar{\alpha}(n_m))) \\ &= v(\bar{\alpha}(f(n_1, \dots, n_m))) \\ &= \tilde{\alpha}(f(n_1, \dots, n_m)), \end{aligned}$$

showing that  $f$  also tracks  $\sigma$  with respect to  $\tilde{\alpha}$ . ■

Note that the numbering  $\tilde{\alpha}$  is induced by the numbering of the representing domain. The effective domain representation of  $A$  is described by the following diagram.



Now we introduce two notions of effectivity for functions between effectively domain representable topological spaces.



**Definition 5.3.5.** Let  $A$  and  $B$  be topological spaces, effectively domain representable by  $(D, D_A, v_A, \alpha)$  and  $(E, E_B, v_B, \beta)$ , respectively.

1. A continuous function  $f : A \rightarrow B$  is said to be  $(\alpha, \beta)$ -effective if there is an  $(\alpha, \beta)$ -effective continuous function  $\bar{f} : D \rightarrow E$  representing  $f$ , that is  $\bar{f}[D_A] \subseteq E_B$  and for each  $x \in D_A$ ,  $f(v_A(x)) = v_B(\bar{f}(x))$ .
2. A function  $f : A_k \rightarrow B_k$  is  $(\tilde{\alpha}, \tilde{\beta})$ -computable, where  $\tilde{\alpha}$  and  $\tilde{\beta}$  are the numberings obtained in Proposition 5.3.4, if there is a partial recursive function  $\hat{f}$  such that  $\Omega_A \subseteq \text{dom}(\hat{f})$  and for all  $n \in \Omega_A$ ,

$$f(\tilde{\alpha}(n)) = \tilde{\beta}(\hat{f}(n)),$$

that is  $\hat{f}$  tracks  $f$  with respect to  $\tilde{\alpha}$  and  $\tilde{\beta}$ .

We have the following diagrams.

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ v_A \uparrow & & \uparrow v_B \\ D_A & \xrightarrow{\bar{f}|_{D_A}} & E_B \end{array} \quad \begin{array}{ccc} A_k & \xrightarrow{f} & B_k \\ \tilde{\alpha} \uparrow & & \uparrow \tilde{\beta} \\ \Omega_A & \xrightarrow{\hat{f}} & \Omega_B \end{array}$$

Note that in (2) we do not require  $f$  to be continuous. In Section 5.5 we will prove a version of Ceitin's Theorem for certain effectively domain representable spaces, saying that each  $(\tilde{\alpha}, \tilde{\beta})$ -computable function is continuous. In particular, we prove Ceitin's Theorem for the computable reals (defined in 5.3.13 below).

**Construction 5.3.6 (The inverse limit construction).** We are going to study the effectivity of the inverse limit construction as given in Section 4.3. Let  $A = (A; a_1, \dots, a_p, \sigma_1, \dots, \sigma_q)$  be an  $\alpha_0$ -computable  $\Sigma$ -algebra and suppose  $\{\equiv_n\}_{n \in \omega}$  is a separating family of congruences on  $A$  which is  $\alpha_0$ -decidable uniformly in  $n$ . Recall that we let  $A_n = A / \equiv_n$  be the set of equivalence classes of  $\equiv_n$  and that  $\mathcal{C} = \bigcup \{A_n : n \in \omega\}$ , the disjoint union of the  $A_n$ . Denoting the elements of  $\mathcal{C}$  by  $[a]_n$  for  $a \in A$  and  $n \in \omega$ , we have that  $\mathcal{C}$  is a c usl under the order

$$[a]_m \sqsubseteq [b]_n \Leftrightarrow m \leq n \text{ and } a \equiv_m b.$$

Then  $\mathcal{C}$  is an  $\alpha$ -computable c usl where  $\alpha : \omega \rightarrow \mathcal{C}$  is defined by

$$\alpha(\langle m, n \rangle) = [\alpha_0(m)]_n.$$

Here,  $\langle \cdot, \cdot \rangle : \omega^2 \rightarrow \omega$  is a recursive pairing function. Thus  $D(A) = \bar{\mathcal{C}}$ , the ideal completion of  $\mathcal{C}$ , is an  $\alpha$ -effective domain. Furthermore,  $\hat{A} =$

$\varprojlim A_n = D(A)_m$ , the set of maximal elements of  $D(A)$ . Note that  $\hat{A}$  is a total set in  $D(A)$ , in the sense of Definition 4.4.1. We summarise our discussion.

**Theorem 5.3.7.** *If  $A$  is a computable  $\Sigma$ -algebra and  $\{\equiv_n\}_{n \in \omega}$  is a family of separating congruences on  $A$ , decidable uniformly in  $n$ , then  $\hat{A} = \varprojlim A_n$  is effectively domain representable by  $(D(A), D(A)_m, \text{id}, \alpha)$ .*

Let us reconsider Examples in 4.3.2.

**Examples 5.3.8 (Examples using inverse limits).**

1. The family of separating equivalences  $\{\equiv_n\}_{n \in \omega}$  on  $\mathbb{N}$  in 4.3.2 (1) is clearly decidable uniformly in  $n$ . Thus the completion of  $\mathbb{N}$ , i.e. the one-point compactification of  $\mathbb{N}$ , is effectively domain representable. Of course, in this case we can do much better. The one-point compactification of  $\mathbb{N}$  is clearly computable.
2. The term algebra  $T(\Sigma, X)$  of terms over a set of variables  $X$  and a signature  $\Sigma$  is clearly computable (provided  $X$  and  $\Sigma$  are given in a computable way) and the family of separating congruences  $\{\equiv_n\}_{n \in \omega}$  is decidable uniformly in  $n$ . It follows that the algebra  $T^\infty(\Sigma, X)$  of all finite and infinite terms is effectively domain representable.
3. Let  $R$  be a computable local ring with unique maximal ideal  $\mathbf{m}$ . It is shown in Stoltenberg-Hansen and Tucker [1988] that  $R$  has a decidable ideal membership relation. This means that the relation

$$a \in (b_1, \dots, b_n)$$

is decidable uniformly in  $n$ . It follows that the congruence relation  $\equiv_n$  defined by

$$a \equiv_n b \Leftrightarrow a - b \in \mathbf{m}^n$$

is decidable uniformly in  $n$ . Thus the local ring  $\hat{R}$ , the completion of  $R$ , is effectively domain representable. It turns out that  $\hat{R}_k$  is also a local ring.

4. Consider  $2^\omega = \{f \mid f : \omega \rightarrow \{0, 1\}\}$ , the Cantor set. Clearly  $2^\omega$  is not computable. However, let  $SEQ$  be the set of finite sequences of 0s and 1s, and for  $\sigma, \tau \in SEQ$  let  $\sigma \sqsubseteq \tau$  just in case  $\tau$  extends  $\sigma$ . Then clearly  $SEQ = (SEQ; \sqsubseteq, <>)$  is a computable c usl, where  $<>$  is the empty sequence. In addition  $SEQ$  is isomorphic to the c usl obtained from  $2^\omega$  using the family of separating congruences  $\{\equiv_n\}_{n \in \omega}$  defined in 4.3.2 (4). It follows that  $2^\omega$  is effectively domain representable. Note that  $(2^\omega)_k$  is the set of (characteristic functions of the) recursive subsets of  $\omega$ .

**Construction 5.3.9 (Locally compact Hausdorff spaces).** Recall from Section 4.5 that a domain representation of a locally compact Haus-

dorff algebra is obtained by taking the ideal completion  $\overline{P}$  of a csl of compact neighbourhood systems  $P$ . We now consider the effectivity of this construction. In order for the construction to provide an effective domain representation we must require that  $P$  be a computable structure. However, we often require more. For example, it is often convenient and for some applications necessary to be able to go effectively from a computable total ideal in  $\overline{P}$  to the least equivalent total ideal. A sufficient condition for this is that it is semidecidable whether  $F \in P$  is contained in the interior of  $G \in P$ .

**Lemma 5.3.10.** *Let  $P$  be an  $\alpha$ -computable csl of compact neighbourhood systems for a locally compact Hausdorff space. Suppose the relation  $F \subseteq G^O$  is  $\alpha$ -semidecidable for  $F, G \in P$ , where  $G^O$  denotes the interior of  $G$ . Let  $f : \overline{P} \rightarrow \overline{P}$  be the function defined by*

$$f(I) = \{G \in P : (\exists F \in I)(F \subseteq G^O)\}.$$

1.  $f$  is a continuous  $(\alpha, \alpha)$ -effective function and  $f[\overline{P}_t] \subseteq \overline{P}_t$ .
2.  $f$  restricted to  $\overline{P}_k$  is  $\overline{\alpha}$ -computable.

**Proof.** (1) That  $f$  is continuous and takes total ideals to total ideals is similar to the proof of Lemma 4.5.14. The effectivity of  $f$  follows immediately by the added hypothesis. For (2) let  $t : \omega \rightarrow \omega$  be the total recursive function of Lemma 5.2.5. Suppose we are given an  $\overline{\alpha}$ -index  $e$  for the ideal  $I \in \overline{P}_k$ . Then  $t(e)$  is an r.e. index for the set  $\alpha^{-1}[\text{approx}(I)] = \alpha^{-1}[I]$ . But then an r.e. index for the set  $\alpha^{-1}\{\{G \in P : (\exists F \in I)(F \subseteq G^O)\}\}$  is obtained uniformly from  $t(e)$  using the hypothesis, say by a total recursive function  $s$ . Again using Lemma 5.2.5 we see that  $\overline{\alpha}(s(t(e))) = f(I)$ , that is the recursive function  $s \circ t$  tracks  $f$ . ■

**Corollary 5.3.11.** *Let  $P$  be an  $\alpha$ -computable csl of compact neighbourhood systems for a locally compact Hausdorff space  $X$  and suppose the relation  $F \subseteq G^O$  is  $\alpha$ -semidecidable for  $F, G \in P$ . Then  $x \in X_k$  if, and only if, the ideal  $I_x = \{G \in P : x \in G^O\} \in \overline{P}_k$ .*

**Proof.** For the non-trivial direction assume  $x \in X_k$ . Let  $I$  be an  $\alpha$ -computable ideal such that  $\bigcap I = \{x\}$  and let  $f$  be the effective function from Lemma 5.3.10. Then  $f(I) = I_x$  and hence  $I_x$  is computable by the effectivity of  $f$ . ■

Next we give a sufficient condition for continuous functions between locally compact Hausdorff spaces to be effective. The crucial condition is the semidecidability of whether, for given compact sets, the image of one is in the interior of the other.

**Proposition 5.3.12.** *Let  $X$  and  $Y$  be locally compact Hausdorff spaces with csls  $P$  and  $Q$  of compact neighbourhood systems, respectively. Suppose  $P$  is  $\alpha$ -computable and  $Q$  is  $\beta$ -computable. Let  $f : X \rightarrow Y$  be a*

continuous function such that the relation

$$f[F] \subseteq G^O$$

is  $(\alpha, \beta)$ -semidecidable for all (compact)  $F \in P$  and  $G \in Q$ . Then  $f$  is  $(\alpha, \beta)$ -effective.

**Proof.** Let  $\bar{f} : \bar{P} \rightarrow \bar{Q}$  be the continuous function representing  $f$ , defined in the proof of Lemma 4.5.14. Thus, for compact  $F \in P$ ,

$$\bar{f}(F) = \{G \in Q : f[F] \subseteq G^O\},$$

that is

$$[G] \subseteq \bar{f}(F) \Leftrightarrow f[F] \subseteq G^O.$$

This relation is by assumption  $(\alpha, \beta)$ -semidecidable, so  $\bar{f}$  is  $(\alpha, \beta)$ -effective. ■

Now we consider the real numbers  $\mathbb{R}$  and show that  $\mathbb{R}$  is effectively domain representable in such a way that the basic effective notions correspond exactly to the usual ones in recursive analysis. A nice general reference for recursive analysis is Pour-El and Richards [1989]. We use their terminology.

**Construction 5.3.13 (The reals  $\mathbb{R}$ ).** Let  $\mathbb{P} = \{[a, b] : a \leq b, a, b \in \mathbb{Q}\} \cup \{\mathbb{R}\}$  and order  $\mathbb{P}$  by reverse inclusion, i.e.

$$[a, b] \subseteq [c, d] \Leftrightarrow [c, d] \subseteq [a, b] \Leftrightarrow a \leq c \text{ \& } d \leq b,$$

and  $\mathbb{R}$  is the least element in  $\mathbb{P}$ . Then  $\mathbb{P}$  is a cusp of compact neighbourhood systems for  $\mathbb{R}$  as in Definition 4.5.2.

Let  $\alpha_0 : \omega \rightarrow \mathbb{Q}$  be a computable numbering of the ordered field of rational numbers. It is easy to see that  $\mathbb{Q}$  is computably stable, that is up to recursive equivalence there is only one such numbering. Therefore we can, and generally will, suppress the numbering  $\alpha_0$  when discussing computability notions for  $\mathbb{Q}$ . Define  $\alpha : \omega \rightarrow \mathbb{P}$  by  $\alpha(0) = \mathbb{R}$  and

$$\alpha(\langle m, n \rangle + 1) = \begin{cases} [\alpha_0(m), \alpha_0(n)] & \text{if } \alpha_0(m) \leq \alpha_0(n) \\ [\alpha_0(n), \alpha_0(m)] & \text{otherwise.} \end{cases}$$

Then  $\alpha$  is a computable numbering of the cusp  $\mathbb{P}$ . It follows from Theorem 4.5.11 that  $\mathbb{R}$  is effectively domain representable by  $(\bar{\mathbb{P}}, \bar{\mathbb{P}}_t, v_\sim, \alpha)$ . It is this domain representation for  $\mathbb{R}$  that we consider in our discussion below.

**Definition 5.3.14.** An element  $x \in \mathbb{R}$  is a *computable real* if there is a computable sequence of rational numbers  $(r_n)$  such that for each  $n$ ,

$$|r_n - x| < 2^{-n}.$$





**Definition 5.3.16.** A sequence  $(x_n)$  of real numbers is *computable* if there is a computable double sequence  $(r_{nk})$  of rational numbers such that

$$|r_{nk} - x_n| \leq 2^{-k} \quad \text{for all } k \text{ and } n.$$

**Definition 5.3.17.** A function  $f : \mathbb{R} \rightarrow \mathbb{R}$  is *computable* if the following hold.

1. For any computable sequence  $(x_n)$  of real numbers, the sequence  $(f(x_n))$  is computable.
2. There is a recursive function  $d : \omega^2 \rightarrow \omega$  such that for all natural numbers  $N$  and  $M$  and for each  $x, y \in [-M, M]$ ,

$$|x - y| < \frac{1}{d(M, N)} \Rightarrow |f(x) - f(y)| < 2^{-N}.$$

We need to elaborate somewhat on condition (1) in order to make it precise. We mean that there should be a uniform effective procedure which given a computable sequence  $(x_n)$  produces a computable sequence  $(f(x_n))$ . That is, there must exist a partial recursive function which given an index for a computable double sequence of rationals  $(r_{nk})$  for  $(x_n)$  computes an index of a computable double sequence  $(s_{nk})$  for  $(f(x_n))$ . An *index for a computable function*  $f : \mathbb{R} \rightarrow \mathbb{R}$  is then a pair of indices, the first being an index of the recursive function existing by condition (1) and the second an index of the recursive function  $d$  of condition (2).

Condition (2) assures that a computable function on  $\mathbb{R}$  is continuous. Thus, in order to show that such a function  $f$  is effective, it suffices by Proposition 5.3.12 to show that the relation  $f([a, b]) \subseteq [c, d]^O$  is  $\alpha$ -semidecidable, where  $f([a, b])$  denotes the image of  $[a, b]$  under  $f$ . Of course,  $f([a, b])$  is again a compact interval by continuity.

The first step is to show that we can compute maximum and minimum values of computable functions on compact intervals.

**Lemma 5.3.18.** *Let  $f : \mathbb{R} \rightarrow \mathbb{R}$  be a computable function and let  $M = \sup\{f(x) : x \in [a, b]\}$ , where  $a \leq b \in \mathbb{Q}$ . Then  $M$  is a computable real number with an index obtained uniformly from  $a$  and  $b$  and an index of  $f$ .*

**Proof.** If  $a = b$  then  $M$  is trivially computable. So we assume  $a < b$ . We are to find a computable sequence  $(w_n)$  of rational numbers such that for each  $n$ ,

$$|w_n - M| < 2^{-n}.$$

First we compute a natural number  $L$  such that  $[a, b] \subseteq [-L, L]$ . Then we define a recursive function  $k : \omega \rightarrow \omega$  by

$$k(n) = \text{least } m \text{ s.t. } \frac{m}{b-a} > d(L, n)$$

where  $d$  is the computable modulus function for  $f$ . For each  $n$  we partition the interval  $[a, b]$  into  $k(n)$  subintervals, each of length

$$\delta(n) = \frac{b-a}{k(n)}.$$

For  $i = 1, \dots, k(n)$  let

$$r_{ni} = a + (i - 1/2)\delta(n).$$

It follows that for each  $i$ ,

$$x \in [a + (i - 1)\delta(n), a + i\delta(n)] \Rightarrow |f(x) - f(r_{ni})| < 2^{-n}.$$

The sequence  $(r_{ni})$  of rational numbers, for  $i = 1, \dots, k(n)$  and  $n = 1, 2, \dots$ , is computable and hence the sequence  $(f(r_{ni}))$  is a computable sequence of real numbers. Thus there is a computable sequence  $(s_{nij})$  of rational numbers,  $i = 1, \dots, k(n)$  and  $n, j \geq 1$ , such that for each such  $n, i, j$ ,

$$|s_{nij} - f(r_{ni})| < 2^{-j}.$$

Now we define  $w_n$  by

$$w_n = \max\{s_{nin} : i = 1, \dots, k(n)\}.$$

Thus  $(w_n)$  is a computable sequence of rational numbers.

Let  $x_0 \in [a, b]$  be such that  $f(x_0) = M$ . (Note that we cannot necessarily compute  $x_0$  but we know that such  $x_0$  exists.) Given  $n$  let  $i$  be such that  $x_0 \in [a + (i - 1)\delta(n), a + i\delta(n)]$ . Then

$$\begin{aligned} |M - s_{nin}| &= |f(x_0) - s_{nin}| \\ &\leq |f(x_0) - f(r_{ni})| + |f(r_{ni}) - s_{nin}| \\ &< 2^{-n} + 2^{-n} = 2^{-n+1}. \end{aligned}$$

Now let  $j$  be such that  $w_n = s_{njn}$ . Then  $f(r_{nj}) \leq M$  and  $s_{nin} \leq w_n$ . In case  $M \leq w_n$ , then

$$|M - w_n| \leq |f(r_{nj}) - w_n| < 2^{-n}.$$

On the other hand, if  $w_n \leq M$ , then

$$|M - w_n| \leq |M - s_{nin+1}| < 2^{-n+1}.$$

In either case,  $|M - w_n| < 2^{-n+1}$ . ■

**Lemma 5.3.19.** *Let  $f : \mathbb{R} \rightarrow \mathbb{R}$  be a computable function. Then the relation*

$$f([a, b]) \subseteq [c, d]^O$$

is semidecidable.

**Proof.** Suppose we are given  $[a, b], [c, d] \in \mathbb{P}$ . We know that  $f([a, b]) = [m, M]$  and by Lemma 5.3.18 we can compute indices for  $m$  and  $M$  as computable real numbers. Thus it suffices to show that the relations  $c < m$  and  $M < d$  are semidecidable. Let  $(r_n)$  be the computable sequence of rational numbers determining  $M$ . Then, clearly,

$$M < d \Leftrightarrow (\exists n)(r_n + 2^{-n} < d)$$

and hence the relation is semidecidable. The case for  $m$  is similar. ■

**Corollary 5.3.20.** *Each computable function  $f : \mathbb{R} \rightarrow \mathbb{R}$  is effective.*

**Proof.** By Proposition 5.3.12 and Lemma 5.3.19. ■

The converse of the above corollary also holds which we now set out to prove. First we observe the following general fact.

**Lemma 5.3.21.** *Let  $f : X \rightarrow Y$  be a continuous function between locally compact Hausdorff spaces and let  $f$  be represented by  $\bar{f} : \bar{P} \rightarrow \bar{Q}$ , where  $P$  and  $Q$  are cusls of compact neighbourhood systems for  $X$  and  $Y$  respectively. Then, for  $F \in P$  and  $G \in Q$ ,*

$$G \in \bar{f}(F) \Rightarrow f[F] \subseteq G.$$

**Proof.** Assume  $G \in \bar{f}(F)$  and let  $x \in F$ . For the maximal ideal  $I^x$  representing  $x$  we have by continuity

$$\bar{f}(I^x) = \bigcup \{\bar{f}(F) : F \in I^x\} = \bigcup \{\bar{f}(F) : x \in F\}.$$

On the other hand, since  $\bar{f}$  represents  $f$ ,

$$\bar{f}(I^x) \subseteq i^{f(x)} = \{G \in Q : f(x) \in G\}.$$

This shows that  $f(x) \in G$ . ■

Now assume that  $f : \mathbb{R} \rightarrow \mathbb{R}$  is effective. Thus  $f$  is represented by an effective continuous function  $\bar{f} : \bar{\mathbb{P}} \rightarrow \bar{\mathbb{P}}$ . By Lemma 5.3.10 we may assume that  $\bar{f}(I) = I_{f(x)}$  whenever  $I$  is a total ideal representing  $x$ . We shall show that  $f$  is computable.

To prove condition (1) let  $(x_n)$  be a computable sequence of reals and let  $(w^{nk})$  be an associated computable double sequence of rationals such

that  $|w_{nk} - x_n| < 2^{-k}$  for each  $n$  and  $k$ . Then the ideal  $I_{x_n}$  is computable, uniformly in  $n$ , since it is generated by

$$[w_{nk} - 2^{-k}, w_{nk} + 2^{-k}] \text{ for } k \in \omega.$$

By the effectivity of  $\bar{f}$ , the ideal  $I_{f(x_n)} = \bar{f}(I_{x_n})$  is computable, uniformly in  $n$ . Thus, given a number  $t$ , we search effectively for  $[c, d] \in I_{f(x_n)}$  such that  $|d - c| < 2^{-t}$ . In this way we obtain a computable double sequence  $(s_{nt})$  by letting

$$s_{nt} = (c + d)/2.$$

Clearly  $|s_{nt} - f(x_n)| < 2^{-t}$  for each  $n$  and  $t$ , so  $(f(x_n))$  is a computable sequence.

It remains to prove the existence of a recursive modulus function for  $f$ . Suppose we are given positive natural numbers  $M$  and  $N$ . For each  $k \in \omega$ , partition  $[-M, M]$  into  $k$  subintervals

$$[a_{ki}, a_{ki+1}] \quad i = 0, \dots, k-1$$

of equal length.

**Claim.** There is  $k$  such that for each  $i = 0, \dots, k-1$  there is  $[c, d] \in \bar{f}([a_{ki}, a_{ki+1}])$  such that  $|d - c| < s^{-N}$ .

Suppose the claim is false for  $N$ . Then for each  $k$  we choose  $i_k$ ,  $0 \leq i_k \leq k-1$ , such that

$$[c, d] \in \bar{f}([a_{ki_k}, a_{ki_k+1}]) \Rightarrow |d - c| \geq 2^{-N}.$$

Choose  $x_k \in [a_{ki_k}, a_{ki_k+1}]$ . Then, by compactness, the set  $\{x_k : k \in \omega\}$  has a limit point  $x$  in  $[-M, M]$ . Suppose first that  $x \neq \pm M$ . Choose  $[c, d] \in I_{f(x)} = \bar{f}(I_x)$  such that  $|d - c| < w^{-N}$ . Thus for some  $[a, b] \in I_x$ ,  $[c, d] \in \bar{f}([a, b])$ . But then there is  $k$  sufficiently large so that  $[a_{ki_k}, a_{ki_k+1}] \subseteq [a, b]$  and hence

$$[c, d] \in \bar{f}([a_{ki_k}, a_{ki_k+1}])$$

which is a contradiction. In case  $x = M$  we consider the ideal  $I_x^L$ , generated by  $\{[a, M] : a < M\}$ , in place of  $I_x$  to obtain a contradiction. The case  $x = -M$  is handled similarly, completing the proof of the claim.

Given  $M$  and  $N$  we compute a number  $k$  which witnesses the claim. The computation is performed by an effective search, using the effectivity of  $\bar{f}$ . Then we define  $d : \omega^2 \rightarrow \omega$  by

$$d(M, N) = \text{least number } > k/2M.$$

Thus  $d$  is a total recursive function with an index obtained uniformly from an index of  $\bar{f}$ .

Suppose  $x, y \in [-M, M]$  and  $|x - y| < 1/d(M, N)$ . Then for the  $k$  computed above,  $|x - y| < 2M/k$  so there is  $i$  such that

$$|x - a_{ki}| < 2M/k \quad \text{and} \quad |y - a_{ki}| < 2M/k.$$

Say, without loss of generality, that  $x \in [a_{ki}, a_{ki+1}]$ . Then there is  $[c, d] \in \bar{f}([a_{ki}, a_{ki+1}])$  such that  $|d - c| < 2^{-N}$ . By Lemma 5.3.21

$$f[a_{ki}, a_{ki+1}] \subseteq [c, d]$$

so  $|f(x) - f(a_{ki})| < 2^{-N}$ . The same holds for  $y$  and hence we have

$$|f(x) - f(y)| \leq |f(x) - f(a_{ki})| + |f(a_{ki}) - f(y)| < 2^{-N} + 2^{-N} = 2^{-N+1}.$$

We have proved the following theorem.

**Theorem 5.3.22.** *A function  $f : \mathbb{R} \rightarrow \mathbb{R}$  is computable in the sense of recursive analysis if, and only if,  $f$  is effective in the sense of Definition 5.3.5.*

## 5.4 The Myhill–Shepherdson Theorem

The Myhill–Shepherdson Theorem states that the effective operators on  $\mathcal{PR}$ , the partial recursive functions on  $\mathbb{N}$ , are exactly the restrictions of the effectively continuous operators on  $\mathcal{P}$ , the partial functions on  $\mathbb{N}$ . Let  $\mathcal{P} = (\mathcal{P}, \alpha)$  be the effective domain of partial functions where  $\alpha$  is a standard numbering. Then  $\mathcal{PR} = \mathcal{P}_k$  and the Myhill–Shepherdson Theorem translated into our language reads that the  $(\bar{\alpha}, \bar{\alpha})$ -computable functions  $f : \mathcal{P}_k \rightarrow \mathcal{P}_k$  are exactly the restrictions to  $\mathcal{P}_k$  of the effective continuous functions from  $\mathcal{P}$  into  $\mathcal{P}$ , that is the restrictions to  $\mathcal{P}_k$  of the elements in  $[\mathcal{P} \rightarrow \mathcal{P}]_k$ . Recall from Section 5.2 that  $\bar{\alpha}$  is the canonical numbering of the computable elements obtained from  $\alpha$ .

The Rice–Shapiro Theorem is closely related to that of Myhill–Shepherdson. It states that a class  $\mathcal{A}$  of r.e. sets is completely r.e. if and only if there is an r.e. set  $V$  such that for each r.e. set  $W$ ,

$$W \in \mathcal{A} \Leftrightarrow (\exists e \in V)(D_e \subseteq W),$$

where  $D_e$  is the finite set with canonical index  $e$ . That  $\mathcal{A}$  is *completely r.e.* means that the set  $\{e \in \omega : W_e \in \mathcal{A}\}$  is r.e. Thus the Rice–Shapiro Theorem in our setting reads as follows. Let  $\wp(\omega) = (\wp(\omega), \alpha)$  be the effective domain of subsets of  $\omega$ . Then  $\mathcal{A} \subseteq \wp(\omega)_k$  is  $\bar{\alpha}$ -semidecidable if and only if there is an r.e. set  $V$  such that

$$\bar{\alpha}(n) \in \mathcal{A} \Leftrightarrow (\exists e \in V)(\alpha(e) \subseteq \bar{\alpha}(n)).$$

In this section we extend the Rice–Shapiro Theorem to an arbitrary



effective domain. Then we show that this theorem also is a generalisation of the Myhill–Shepherdson Theorem.

Recall that the Scott topology on a domain  $D$  is generated by the basic open sets  $B_a = \{x \in D : a \sqsubseteq x\}$  for  $a \in D_c$ .

**Definition 5.4.1.** Let  $(D, \alpha)$  be an effective domain.

1. A set  $U \subseteq D$  is *effectively open* if there is an r.e. set  $W$  such that

$$U = \bigcup_{e \in W} B_{\alpha(e)}.$$

An r.e. index of  $W$  is an *index* of the effectively open set  $U$ .

2. If  $B \subseteq D$  then  $U \subseteq B$  is *effectively open* in  $B$  if there is an effectively open set  $V \subseteq D$  such that  $U = V \cap B$ . An index of  $V$  is also an index of  $U$ .

In particular, an effective open set is open in the (relativised) Scott topology. Here is the main theorem.

**Theorem 5.4.2.** (Ershov[1977a]). Let  $(D, \alpha)$  be an effective domain. Then  $U \subseteq D_k$  is  $\bar{\alpha}$ -semidecidable if, and only if,  $U$  is effectively open in  $D_k$ .

The proof we give uses the following recursion-theoretic lemma called the Branching Lemma.

**Lemma 5.4.3.** (Berger [1993]). Let  $V$  and  $W$  be r.e. sets such that  $W$  contains all r.e. indices of  $V$ . Let  $\lambda p.V^p$  be an enumeration of  $V$  and let  $r$  be a total recursive function. Then there is  $e \in W$  and  $p \in \mathbb{N}$  such that

$$W_e = V^p \cup W_{r(p)}.$$

Furthermore, such  $e$  and  $p$  are computed uniformly from an r.e. index of  $W$  and recursive indices of the functions  $\lambda p.V^p$  and  $r$ .

**Proof.** Let  $\lambda p.W^p$  be an enumeration of  $W$  such that  $W^0 = \emptyset$  and define  $W_e$ , using the Second Recursion Theorem, by

$$x \in W_e \Leftrightarrow \exists p((x \in V^p \ \& \ e \notin W^p) \vee (x \in W_{r(p)} \ \& \ e \in W^{p+1} - W^p)).$$

If  $e \notin W$  then  $W_e = V$  by the definition of  $W_e$ , so  $e \in W$  by the assumption on  $W$ . It follows that  $e \in W$ . Let  $p$  be the unique number such that  $e \in W^{p+1} - W^p$ . Then

$$W_e = V^p \cup W_{r(p)}.$$

The claimed uniformities follow from the uniformity of the Second Recursion Theorem. ■

The lemma says that if  $W$  contains all r.e. indices of  $V$  then, for some  $p$ , it also contains an r.e. index of  $W_{r(p)}$  modulo a finite part of  $V$ . That is, we branch along  $W_{r(p)}$  for some  $p$  and still stay within  $W$ .

**Proof.** [Proof of 5.4.2] First assume that  $U$  is effectively open in  $D_k$ . Let  $W$  be an r.e. set such that  $U = \bigcup_{e \in W} (B_{\alpha(e)} \cap D_k)$ . Then

$$\bar{\alpha}(n) \in U \Leftrightarrow (\exists e \in W)(\alpha(e) \sqsubseteq \bar{\alpha}(n))$$

which is an r.e. relation since  $\bar{\alpha}$  is a constructive numbering. Thus  $U$  is  $\bar{\alpha}$ -semidecidable.

For the converse, assume  $U \subseteq D_k$  is  $\bar{\alpha}$ -semidecidable. Then the set  $W = \{e \in \omega : \alpha(e) \in U\}$  is r.e. since the inclusion  $\iota : D_c \rightarrow D_k$  is  $(\alpha, \bar{\alpha})$ -computable. Suppose we have shown that  $U$  is an open set. Then, by the Alexandrov and Scott conditions on open sets,

$$U = \bigcup_{e \in W} (B_{\alpha(e)} \cap D_k)$$

so that  $U$  is effectively open.

It thus remains to prove that  $U$  is open; that is, that it satisfies the Alexandrov and Scott conditions. We first consider the former. Let  $x = \bar{\alpha}(m)$  and  $y = \bar{\alpha}(n)$  and suppose that  $x \in U$  and  $x \sqsubseteq y$ . We shall show  $y \in U$ , using Lemma 5.4.3. Let  $W = \bar{\alpha}^{-1}[U]$  and let  $t$  be the total recursive function of Lemma 5.2.5. If  $W_e = W_{t(m)}$  then, by Lemma 5.2.5,

$$\bar{\alpha}(e) = \bar{\alpha}(t(m)) = \bar{\alpha}(m) = x \in U$$

so that  $W$  contains all r.e.-indices of  $W_{t(m)}$ . By Lemma 5.4.3, letting  $r$  be the constant function with value  $t(n)$ , there is  $p \in \omega$  and  $e \in W$  such that

$$W_e = W_{t(m)}^p \cup W_{t(n)}.$$

But  $W_{t(m)} = \alpha^{-1}[\text{approx}(x)] \subseteq \alpha^{-1}[\text{approx}(y)] = W_{t(n)}$  and hence  $W_e = W_{t(n)}$ . Thus, again by Lemma 5.2.5,  $y = \bar{\alpha}(n) = \bar{\alpha}(t(n)) = \bar{\alpha}(e) \in U$ .

To verify the Scott condition let  $x, m$ , and  $W$  be as above. We define a total recursive function  $r$  such that  $W_{r(p)}$  is a singleton set consisting of an  $\alpha$ -index for  $\bigsqcup \alpha[W_{t(m)}^p]$ . Again by Lemma 5.4.3 there is  $p \in \omega$  and  $e \in W$  such that  $W_e = W_{t(m)}^p \cup W_{r(p)}$ . Thus  $\alpha[W_e]$  is directed and hence, by Lemma 5.2.5,

$$\bar{\alpha}(e) = \bigsqcup \alpha[W_e] = \bigsqcup \alpha[W_{t(m)}^p \cup W_{r(p)}] \sqsubseteq \bar{\alpha}(t(m)) = x.$$

Furthermore,  $\bar{\alpha}(e)$  is compact since  $\bigsqcup \alpha[W_{t(m)}^p \cup W_{r(p)}]$  is the supremum of a finite set of compact elements. ■

The theorem is uniform in the sense that there are total recursive functions which given an  $\bar{\alpha}$ -semidecidable index of  $U$  compute an index of  $U$  as an effectively open set and conversely.

To say that a class  $\mathcal{A}$  of r.e. sets is completely r.e. means precisely that  $\mathcal{A}$  is an  $\bar{\alpha}$ -semidecidable subset of  $\wp(\omega)_k$ . And to say that  $\mathcal{A}$  is effectively open in  $\wp(\omega)_k$  means that there is an r.e. set  $V$  such that

$$W \in \mathcal{A} \Leftrightarrow (\exists e \in V)(\alpha(e) \sqsubseteq_{\wp(\omega)} W)$$

that is

$$W \in \mathcal{A} \Leftrightarrow (\exists e \in V)(D_e \subseteq W),$$

when the numbering  $\alpha$  is given by  $\alpha(e) = D_e$ . Thus Ershov's Theorem applied to  $\wp(\omega)$  gives the Rice-Shapiro result.

Ershov's Theorem also provides a generalisation of Rice's Theorem, the latter stating that a class of r.e. sets  $\mathcal{A}$  is *completely recursive* (in our language  $\bar{\alpha}$ -decidable in  $\wp(\omega)_k$ ) if and only if  $\mathcal{A} = \emptyset$  or  $\mathcal{A}$  contains all r.e. sets.

**Corollary 5.4.4.** *Let  $(D, \alpha)$  be an effective domain and suppose  $\mathcal{A} \subseteq D_k$ . Then  $\mathcal{A}$  is  $\bar{\alpha}$ -decidable if and only if  $\mathcal{A} = \emptyset$  or  $\mathcal{A} = D_k$ .*

**Proof.** For the non-trivial direction suppose  $\mathcal{A}$  is  $\bar{\alpha}$ -decidable. Then  $\mathcal{A}$  and  $D_k - \mathcal{A}$  are  $\bar{\alpha}$ -semidecidable and hence, by Ershov's Theorem, open. Using the Alexandrov condition on open sets we see that if  $\perp \in \mathcal{A}$  then  $\mathcal{A} = D_k$  and if  $\perp \notin \mathcal{A}$  then  $D_k - \mathcal{A} = D_k$ , that is  $\mathcal{A} = \emptyset$ . ■

We now turn to the Myhill-Shepherdson Theorem.

**Definition 5.4.5.** Let  $(D, \alpha)$  and  $(E, \beta)$  be effective domains and suppose  $B \subseteq D$  and  $C \subseteq E$ . Then  $f : B \rightarrow C$  is *effectively continuous* if there is a total recursive function  $g$  such that  $f^{-1}[B_{\beta(e)} \cap C] = \bigcup_{m \in W_{g(e)}} (B_{\alpha(m)} \cap B)$ .

Thus a function  $f : D \rightarrow E$  is effectively continuous if  $f$  is continuous with respect to the Scott topologies and the inverse image of an effectively open set is effectively open uniformly in an index for the effectively open set.

The following proposition is obvious.

**Proposition 5.4.6.** *Let  $(D, \alpha)$  and  $(E, \beta)$  be effective domains and suppose  $D_c \subseteq B \subseteq D$  and  $E_c \subseteq C \subseteq E$ .*

1. *A continuous function  $f : B \rightarrow C$  is effectively continuous if, and only if, the relation  $b \sqsubseteq f(a)$  on  $D_c \times E_c$  is  $(\alpha, \beta)$ -semidecidable.*
2. *A continuous function  $f : B \rightarrow C$  has a unique continuous extension  $\bar{f} : D \rightarrow E$  and  $\bar{f}$  is effective if, and only if,  $f$  is effectively continuous.*

Here is the generalisation of the Myhill-Shepherdson Theorem.

**Theorem 5.4.7.** *Let  $(D, \alpha)$  and  $(E, \beta)$  be effective domains. Then a function  $f : D_k \rightarrow E_k$  is  $(\bar{\alpha}, \bar{\beta})$ -computable if, and only if, there is a continuous  $(\alpha, \beta)$ -effective function  $\bar{f} : D \rightarrow E$  such that  $\bar{f}|_{D_k} = f$ .*

**Proof.** The if direction is Proposition 5.2.13. For the only if direction suppose that  $f : D_k \rightarrow E_k$  is  $(\bar{\alpha}, \bar{\beta})$ -computable and let  $\hat{f}$  be a tracking function of  $f$ . Consider a basic open set  $B_{\beta(e)} \cap E_k$ . Then

$$\bar{\alpha}(n) \in f^{-1}[B_{\beta(e)} \cap E_k] \Leftrightarrow \beta(e) \sqsubseteq f\bar{\alpha}(n) \Leftrightarrow \beta(e) \sqsubseteq \bar{\beta}\hat{f}(n)$$

and the latter relation is r.e. since  $(E_k, \bar{\beta})$  is a constructive subdomain. It follows that  $f^{-1}[B_{\beta(e)} \cap E_k]$  is  $\bar{\alpha}$ -semidecidable with an index obtained uniformly from  $e$ . Then, by Theorem 5.4.2 and its uniformity,  $f^{-1}[B_{\beta(e)} \cap E_k]$  is effectively open with an index obtained uniformly from  $e$ , that is  $f$  is effectively continuous. By Proposition 5.4.6,  $f$  has an extension  $\bar{f} : D \rightarrow E$  which is  $(\alpha, \beta)$ -effective and continuous. ■

## 5.5 The Kreisel–Lacombe–Shoenfield Theorem

In this section we prove a generalisation of the Kreisel–Lacombe–Shoenfield Theorem to effective domains. Then, to illustrate the strength of this generalisation, we apply the theorem to obtain a version of Ceitin’s Theorem using effective domain representability.

In analogy with the Myhill–Shepherdson Theorem, the Kreisel–Lacombe–Shoenfield Theorem also concerns the extension of effective operators to effectively continuous functions. An important difference, which makes the proof more complicated, and interesting, is that in the Kreisel–Lacombe–Shoenfield Theorem the set on which the effective operator is defined is not even semidecidable. Instead the sets have other properties which are abstracted in the generalisation to domains.

Let  $\mathcal{R}$  be the class of total recursive functions on  $\mathbb{N}$ , let  $\mathcal{PR}$  be the class of partial recursive functions, and let  $\mathcal{P}$  be the class of all partial functions on  $\mathbb{N}$ . So  $\mathcal{R} \subseteq \mathcal{PR} \subseteq \mathcal{P}$ .

**Theorem 5.5.1.** *(Kreisel–Lacombe–Shoenfield [1959]). The effective operations on  $\mathcal{R}$  are precisely the restrictions of the effectively continuous functionals on  $\mathcal{P}$  mapping  $\mathcal{R}$  into  $\mathcal{R}$ .*

The significant properties of  $\mathcal{R}$  in the Kreisel–Lacombe–Shoenfield Theorem are that each element in  $\mathcal{R}$  is maximal and hence total in  $\mathcal{P}$  and that  $\mathcal{R}$  is dense in  $\mathcal{P}$  in an effective way. The notion of totality we consider is the one given in Definition 4.4.1. By effectively dense we mean the following.

**Definition 5.5.2.** Let  $(D, \alpha)$  be an effective domain. Then  $M \subseteq D_k$  is *effectively dense* in  $D$  if there is a total recursive function  $d$  such that for each  $n \in \omega$ ,

$$\bar{\alpha}d(n) \in B_{\alpha(n)} \cap M,$$

that is  $\alpha(n) \sqsubseteq \bar{\alpha}d(n) \in M$ .

We now state the generalisation of the Kreisel–Lacombe–Shoenfield Theorem to effective domains. Another generalisation of a more topological nature, to countable  $T_0$ -spaces satisfying certain effectivity requirements, is given in Spreen and Young [1984].

**Theorem 5.5.3.** (Berger [1993]). *Let  $(D, \alpha)$  and  $(E, \beta)$  be effective domains and let  $M \subseteq D_k$  be effectively dense in  $D$ . If  $F : M \rightarrow E_k$  is an  $(\bar{\alpha}, \bar{\beta})$ -computable function such that  $F(x)$  is total in  $E$  for each  $x \in M$  then there is an  $(\bar{\alpha}, \bar{\beta})$ -computable function  $G : D_k \rightarrow E_k$  such that  $F(x) \sqsubseteq G(x)$  for each  $x \in M$ .*

**Remarks 5.5.4.**

1. If  $F(x)$  is maximal in  $E$  for each  $x \in M$  then we obtain equality between  $F$  and  $G$  on  $M$ , that is  $G$  is an extension of  $F$ .
2. The classical Kreisel–Lacombe–Shoenfield Theorem is obtained by letting  $D = E = \mathcal{P}$  and  $M = \mathcal{R}$  using a standard numbering.
3. The function  $G$  can be extended to a continuous effective function  $\bar{G} : D \rightarrow E$  by the Myhill–Shepherdson Theorem.

In the proof we use the following technical lemma, a version of Lemma 5.4.3 where the uniformities are made explicit.

**Lemma 5.5.5.** *Let  $s, t : \mathbb{N} \rightarrow \mathbb{N}$  and  $r : \mathbb{N}^2 \rightarrow \mathbb{N}$  be total recursive functions. Suppose  $M \subseteq \mathbb{N}$  has the property that if  $n \in M$  then  $W_{s(n)}$  contains all r.e. indices of  $W_{t(n)}$ . Then there is a total recursive function  $b : \mathbb{N} \rightarrow \mathbb{N}$  and a partial recursive function  $p : \mathbb{N} \rightarrow \mathbb{N}$  such that  $M \subseteq \text{dom}(p)$  and for each  $n \in \text{dom}(p)$*

$$W_{b(n)} = W_{t(n)}^{p(n)} \cup W_{r(n, p(n))}.$$

**Proof.** Define by the uniform version of the Second Recursion Theorem a total recursive function  $b$  such that

$$\begin{aligned} x \in W_{b(n)} &\Leftrightarrow \exists m((x \in W_{t(n)}^m \ \& \ b(n) \notin W_{s(n)}^m) \vee \\ &(x \in W_{r(n, m)} \ \& \ b(n) \in W_{s(n)}^{m+1} - W_{s(n)}^m)). \end{aligned} \quad (*)$$

Let  $p$  be the partial recursive function defined by

$$p(n) \simeq \mu m[b(n) \in W_{s(n)}^{m+1} - W_{s(n)}^m].$$

From (\*) it then follows that if  $n \in \text{dom}(p)$  then

$$W_{b(n)} = W_{t(n)}^{p(n)} \cup W_{r(n, p(n))}.$$



To show that  $M \subseteq \text{dom}(p)$ , let  $n \in M$ . If  $b(n) \notin W_{s(n)}$  then, again by (\*),  $W_{b(n)} = W_{i(n)}$  and hence, by the assumption on  $M$ ,  $b(n) \in W_{s(n)}$ . Thus  $b(n) \in W_{s(n)}$ , that is  $n \in \text{dom}(p)$ . ■

**Proof.** [Proof of 5.5.3] Let  $\hat{F}$  be a partial recursive function tracking  $F$  with respect to  $(\bar{\alpha}, \bar{\beta})$  and let  $d$  be a total recursive function witnessing that  $M$  is effectively dense in  $D$  with respect to  $\alpha$ .

Recall our approximation functions of Lemma 5.2.6. We have for each  $n \in \bar{\alpha}^{-1}[M]$ ,

$$F(\bar{\alpha}(n)) = \bigsqcup_{i \in \omega} \tilde{\beta}(\hat{F}(n), i).$$

Given  $x \in D_k$  we want to define a set

$$A_x = \{\tilde{\beta}(\hat{F}(n), i) : \langle n, i \rangle \in V_x\}$$

where  $V_x$  is an r.e. set with an index obtained uniformly from an  $\bar{\alpha}$ -index of  $x$ . The intention is to set  $G(x) = \bigsqcup A_x$ . Thus we need to have that  $A_x$  is a consistent set and that  $V_x$  is sufficiently large to ensure  $F(x) \sqsubseteq G(x)$  whenever  $x \in M$ . The latter is achieved by putting  $\langle n, i \rangle \in V_x$  whenever  $\bar{\alpha}(n) = x \in M$ . To show the consistency of  $A_x$  it suffices to consider finite subsets of  $A_x$ . And for this, by Proposition 4.4.4, it will suffice to show that  $\tilde{\beta}(\hat{F}(n), i)$  and  $F(\bar{\alpha}d(r))$  are consistent whenever  $\tilde{\alpha}(n, q) \sqsubseteq \alpha(r)$  for certain  $q$  depending on  $n$  and  $i$ . It is here that the totality of  $F(\bar{\alpha}d(r))$  is used. The strategy is, given  $n, i$ , and  $q$ , to attempt to find an  $r$  such that  $\tilde{\alpha}(n, q) \sqsubseteq \alpha(r)$  and such that  $\tilde{\beta}(\hat{F}(n), i)$  and  $F(\bar{\alpha}d(r))$  are *inconsistent*. By a rather clever use of the Branching Lemma 5.5.5 we obtain a partial recursive function  $p$  such that if  $\tilde{\alpha}(n, p(n, i)) \sqsubseteq \alpha(r)$  then  $\tilde{\beta}(\hat{F}(n), i)$  and  $F(\bar{\alpha}d(r))$  are *consistent*. Thus  $p$  lifts us to a level where we fail to find inconsistent  $\tilde{\beta}(\hat{F}(n), i)$  and  $F(\bar{\alpha}d(r))$  provided  $\tilde{\alpha}(n, p(n, i)) \sqsubseteq \alpha(r)$ .

To start the technical construction we define a relation  $R$  on  $\mathbb{N}$  by

$$R(n, q, i, r) \Leftrightarrow \tilde{\alpha}(n, q) \sqsubseteq \alpha(r) \ \& \ n \in \text{dom}(\hat{F}) \ \& \ \neg \text{Cons}(\tilde{\beta}(\hat{F}(n), i), F(\bar{\alpha}d(r))).$$

Then  $R$  is an r.e. relation since

$$\begin{aligned} \neg \text{Cons}(\tilde{\beta}(\hat{F}(n), i), F(\bar{\alpha}d(r))) &\Leftrightarrow (\exists b \in \text{approx}(F(\bar{\alpha}d(r)))) \\ &\quad \neg \text{Cons}(\tilde{\beta}(\hat{F}(n), i), b). \end{aligned}$$

Let  $f : \omega^3 \rightarrow \omega$  be the partial recursive function defined using a selection operator by

$$f(n, q, i) \simeq (\text{some } r)R(n, q, i, r).$$

Thus there is a total recursive function  $s : \omega^3 \rightarrow \omega$  such that

$$W_{s(n, q, i)} = \begin{cases} \{r\} & \text{if } f(n, q, i) \simeq r \\ \emptyset & \text{if } f(n, q, i) \uparrow. \end{cases}$$

Let  $h : \omega^3 \rightarrow \omega$  be a total recursive function such that

$$W_{h(n,q,i)} = \alpha^{-1}[\text{approx}(\tilde{\alpha}(n,q))] \cup \{k \in \omega : (\exists r \in W_{s(n,q,i)})(\alpha(k) \subseteq \bar{\alpha}d(r))\}.$$

By Lemma 5.2.5 we then have

$$\bar{\alpha}h(n,q,i) = \begin{cases} \bar{\alpha}d(r) & \text{if } W_{s(n,q,i)} = \{r\} \\ \tilde{\alpha}(n,q) & \text{if } W_{s(n,q,i)} = \emptyset. \end{cases}$$

Finally, let  $v : \omega^2 \rightarrow \omega$  be a total recursive function such that

$$W_{v(n,i)} = \{e \in \omega : e \in \text{dom}(\hat{F}) \ \& \ \tilde{\beta}(\hat{F}(n), i) \subseteq \bar{\beta}\hat{F}(e)\}.$$

Suppose  $n \in \bar{\alpha}^{-1}[M]$ ,  $i \in \omega$ , and suppose further that  $W_e = W_{t(n)}$  where  $t$  is the total recursive function of Lemma 5.2.5. Then

$$\bar{\alpha}(e) = \bar{\alpha}t(n) = \bar{\alpha}(n).$$

In particular,  $e \in \bar{\alpha}^{-1}[M]$  so  $e \in \text{dom}(\hat{F})$ . Furthermore,

$$\tilde{\beta}(\hat{F}(n), i) \subseteq \bar{\beta}\hat{F}(n) = F(\bar{\alpha}(n)) = F(\bar{\alpha}(e)) = \bar{\beta}\hat{F}(e).$$

Thus whenever  $\bar{\alpha}(n) \in M$  then each r.e. index of  $W_{t(n)}$  belongs to  $W_{v(n,i)}$ .

By the Branching Lemma 5.5.5 there is a total recursive function  $b : \omega^2 \rightarrow \omega$  and a partial recursive function  $p : \omega^2 \rightarrow \omega$  such that for each  $n, i \in \omega$ , if  $\bar{\alpha}(n) \in M$  then  $p(n, i) \downarrow$  and whenever  $p(n, i) \downarrow$  then  $b(n, i) \in W_{v(n,i)}$  and

$$W_{b(n,i)} = W_{t(n)}^{p(n,i)} \cup W_{h(n,p(n,i),i)}.$$

It is convenient to have chosen the effective enumeration of  $W_{t(n)}$  appearing above given by

$$W_{t(n)}^m = \{k \leq m : \alpha(k) \subseteq \tilde{\alpha}(n, m)\}.$$

For each  $x \in D_k$  let

$$V_x = \{\langle n, i \rangle : (n, i) \in \text{dom}(p), n \in \text{dom}(\hat{F}) \text{ and } \tilde{\alpha}(n, p(n, i)) \subseteq x\}.$$

Then  $V_x$  is r.e. with an index obtained uniformly from an  $\bar{\alpha}$ -index of  $x$ . Finally put

$$A_x = \{\tilde{\beta}(\hat{F}(n), i) : \langle n, i \rangle \in V_x\}.$$

Suppose we have shown that  $A_x$  is consistent. Then we define  $G : D_k \rightarrow E_k$  by

$$G(x) = \bigsqcup A_x.$$

Clearly,

$$\beta(m) \in \text{approx}(G(x)) \Leftrightarrow (\exists k)(\exists \langle n_1, i_1 \rangle, \dots, \langle n_k, i_k \rangle \in V_x) \\ \left( \beta(m) \subseteq \bigsqcup_{j=1}^k \tilde{\beta}(\hat{F}(n_j), i_j) \right).$$

It follows that an r.e. index for  $\beta^{-1}[\text{approx}(G(x))]$  is obtained uniformly from an  $\bar{\alpha}$ -index for  $x$ . Thus, by Lemma 5.2.5, there is a total recursive function  $\hat{G}$  which given an  $\bar{\alpha}$ -index for  $x$  computes a  $\bar{\beta}$ -index for  $G(x)$ . In other words,  $G$  is  $(\bar{\alpha}, \bar{\beta})$ -computable.

Now suppose  $\bar{\alpha}(n) = x \in M$ . Then  $\langle n, i \rangle \in V_x$  and hence  $\tilde{\beta}(\hat{F}(n), i) \in A_x$ . Furthermore

$$F(x) = F(\bar{\alpha}(n)) = \bar{\beta}\hat{F}(n) = \bigsqcup_{i \in \omega} \tilde{\beta}(\hat{F}(n), i)$$

so  $F(x) \subseteq G(x)$ .

It remains to show that  $A_x$  is consistent. A first observation is

$$p(n, i) \downarrow \Rightarrow \bar{\alpha}b(n, i) = \bar{\alpha}h(n, p(n, i), i). \quad (*)$$

For in case the premise holds then, by our choice of enumeration of  $W_{t(n)}, W_{t(n)}^{p(n, i)} \subseteq W_{h(n, p(n, i), i)}$ , so  $W_{b(n, i)} = W_{h(n, p(n, i), i)}$  and hence by Lemma 5.2.5,  $\bar{\alpha}b(n, i) = \bar{\alpha}h(n, p(n, i), i)$ .

Let  $\langle n, i \rangle \in V_x$  and suppose  $\bar{\alpha}(n, p(n, i)) \subseteq \alpha(r)$ . We shall show that  $\tilde{\beta}(\hat{F}(n), i)$  and  $F(\bar{\alpha}d(r))$  are consistent. Suppose not. Then  $R(n, p(n, i), i, r)$  so  $W_{s(n, p(n, i), i)} = \{r'\}$  for some  $r'$  such that  $R(n, p(n, i), i, r')$ . By (\*) and the definition of  $h$ ,

$$\bar{\alpha}b(n, i) = \bar{\alpha}h(n, p(n, i), i) = \bar{\alpha}d(r').$$

Furthermore  $b(n, i) \in W_{v(n, i)}$ , since  $\langle n, i \rangle \in V_x$ . But then

$$\tilde{\beta}(\hat{F}(n), i) \subseteq \bar{\beta}\hat{F}b(n, i) = F(\bar{\alpha}b(n, i)) = F(\bar{\alpha}d(r'))$$

showing that  $\tilde{\beta}(\hat{F}(n), i)$  and  $F(\bar{\alpha}d(r'))$  are indeed consistent, contradicting  $R(n, p(n, i), i, r')$ .

To establish the consistency of  $A_x$  it suffices to consider all its finite subsets. So suppose we are given

$$\tilde{\beta}(\hat{F}(n_1), i_1), \dots, \tilde{\beta}(\hat{F}(n_k), i_k) \in A_x.$$

Then  $\bar{\alpha}(n_1, p(n_1, i_1)), \dots, \bar{\alpha}(n_k, p(n_k, i_k))$  are consistent since they are bounded by  $x$ . Furthermore their supremum is compact, so there is an  $r$  such that  $\bar{\alpha}(n_j, p(n_j, i_j)) \subseteq \alpha(r)$  for  $j = 1, \dots, k$ . By the above,  $\tilde{\beta}(\hat{F}(n_j), i_j)$  and  $F(\bar{\alpha}d(r))$  are consistent for each  $j$ . But then  $\tilde{\beta}(\hat{F}(n_1), i_1), \dots, \tilde{\beta}(\hat{F}(n_k), i_k)$  are consistent by Proposition 4.4.4 since  $F(\bar{\alpha}d(r))$  is total in  $E$  by assumption. ■

The following corollaries use the assumptions of Theorem 5.5.3.

**Corollary 5.5.6.** *If  $F : M \rightarrow E_k$  is  $(\bar{\alpha}, \bar{\beta})$ -computable and  $F(x) \in E_m$  for each  $x \in M$  then  $F$  is effectively continuous.*

**Proof.**  $F$  extends to  $(\bar{\alpha}, \bar{\beta})$ -computable  $\tilde{F} : D_k \rightarrow E_k$  which in turn, by the generalised Myhill–Shepherdson Theorem 5.4.7, extends to a continuous effective function  $\bar{F} : D \rightarrow E$ . Then  $\bar{F}|_M$  is effectively continuous. ■

In general it is not possible to obtain  $G$  in the theorem which is equal to  $F$  on  $M$ . However, if  $N \subseteq E_k$  contains only total elements then we are often interested in the quotient structure of total objects  $\tilde{N} = N / \sim$ , with the quotient topology.

**Corollary 5.5.7.** *Suppose  $F : M \rightarrow N$  is  $(\bar{\alpha}, \bar{\beta})$ -computable where  $M \subseteq D_k$  is effectively dense and  $N \subseteq E_k$  contains only total elements. Then  $\tilde{F} : M \rightarrow \tilde{N}$  defined by  $\tilde{F}(x) = [F(x)]$  is continuous.*

**Proof.** Let  $N' = \{z \in E : (\exists y \in N)(y \sqsubseteq z)\}$ . Then  $N'$  contains only total elements. Furthermore  $N'$  and  $\tilde{N}$  are homeomorphic spaces. By Theorem 5.5.3 there is an  $(\bar{\alpha}, \bar{\beta})$ -computable function  $G : D_k \rightarrow E_k$  such that  $F(x) \sqsubseteq G(x)$  for  $x \in M$ . By the generalised Myhill–Shepherdson Theorem,  $G : M \rightarrow N'$  is continuous and hence  $\tilde{G} : M \rightarrow \tilde{N}'$  is continuous, being a composition of  $G$  and the factoring map. Using a homeomorphism  $h : \tilde{N}' \rightarrow \tilde{N}$  we have  $\tilde{F} = h \circ \tilde{G}$ . ■

We conclude by showing that a version of Ceitin's Theorem is a consequence of the generalised Kreisel–Lacombe–Shoenfield Theorem. Recall that Ceitin-like theorems state that, under certain conditions, computability (in the sense of Definition 5.3.5(2)) implies continuity.

**Theorem 5.5.8.** *Let the topological spaces  $X$  and  $Y$  be effectively domain representable by  $(D, D_X, v, \alpha)$  and  $(E, E_Y, w, \beta)$  respectively, where  $D_X$  and  $E_Y$  contain only total elements. Assume that  $D_X \cap D_k$  is effectively dense in  $D$  and that there is a  $\bar{\beta}$ -computable function  $h : E_Y \cap E_k \rightarrow E_Y \cap E_k$  such that for all  $y, y' \in E_Y \cap E_k$ ,*

$$y \sim y' \Rightarrow h(y) = h(y') \sim y.$$

*Then each  $(\tilde{\alpha}, \tilde{\beta})$ -computable function  $f : X_k \rightarrow Y_k$  is continuous.*

**Proof.** By Proposition 4.4.9 it suffices to find a continuous function  $F : D \rightarrow E$  representing  $f$ , that is,  $F[D_X \cap D_k] \subseteq E_Y \cap E_k$  and for  $x \in D_X \cap D_k$ ,

$$f(v(x)) = w(F(x)).$$

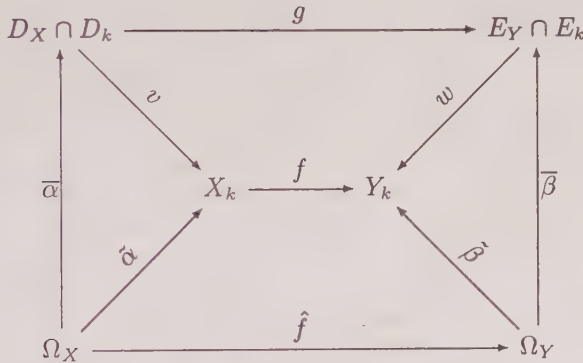
Assume that  $f : X_k \rightarrow Y_k$  is  $(\tilde{\alpha}, \tilde{\beta})$ -computable and let  $\hat{f}$  be a partial recursive function tracking  $f$ . Define  $g : D_X \cap D_k \rightarrow E_Y \cap E_k$  by, for  $n \in \Omega_X$ ,

$$g(\overline{\alpha}(n)) = h(\overline{\beta}\hat{f}(n)).$$

To see that  $g$  is well-defined, assume  $\overline{\alpha}(n) \sim \overline{\alpha}(m)$ . Then  $\tilde{\alpha}(n) = \tilde{\alpha}(m)$  and hence

$$w(\overline{\beta}\hat{f}(n)) = \tilde{\beta}\hat{f}(n) = f\tilde{\alpha}(n) = f\tilde{\alpha}(m) = \tilde{\beta}\hat{f}(m) = w(\overline{\beta}\hat{f}(m)),$$

that is,  $\overline{\beta}\hat{f}(n) \sim \overline{\beta}\hat{f}(m)$ . By the assumptions on  $h$ , we see that  $g$  is well-defined and  $(\overline{\alpha}, \overline{\beta})$ -computable. Consider the diagram below.



To see that the upper rectangle commutes let  $x \in D_X \cap D_k$  and assume  $x = \overline{\alpha}(n)$ . Then

$$f(v(x)) = f(\tilde{\alpha}(n)) = \tilde{\beta}\hat{f}(n) = w(\overline{\beta}\hat{f}(n)) = w(h(\overline{\beta}\hat{f}(n))) = w(g(x)).$$

The conditions of Theorem 5.5.3 are satisfied for  $g$  so we conclude that there is an  $(\overline{\alpha}, \overline{\beta})$ -computable function  $G : D_k \rightarrow E_k$  such that for  $x \in D_X \cap D_k$ ,  $g(x) \subseteq G(x)$ . By the generalised Myhill-Shepherdson Theorem we extend  $G$  to a continuous function  $F : D \rightarrow E$ . Then, for  $x \in D_X \cap D_k$ ,

$$w(F(x)) = w(G(x)) = w(g(x)) = f(v(x))$$

which proves that  $f$  is continuous. ■

Note that the conditions on  $Y$  in the above theorem are trivially satisfied for spaces obtained by an inverse limit as in Theorem 5.3.7. For locally compact Hausdorff spaces, the representation of 5.3.9 satisfies the conditions when the relation  $F \subseteq G^O$  is semidecidable. In particular, we have:



**Corollary 5.5.9.** *If the function  $f : \mathbb{R}_k \rightarrow \mathbb{R}_k$  is computable in the sense of Definition 5.3.5(2) then  $f$  is continuous.*

**Proof.** The required conditions are obviously satisfied for  $\mathbb{R}$  using the standard domain representation of 5.3.13. ■

## References

- [Aberth, 1980] O. Aberth. *Computable Analysis*. McGraw-Hill, New York, 1980.
- [Abramsky and Jung, 1994] S. Abramsky and A. Jung. Domain theory. In S. Abramsky, D. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 3*. Oxford University Press, Oxford, 1994.
- [Adian, 1957] S. I. Adian. The unsolvability of certain algorithmic problems in the theory of groups. *Trudy Moskov Mat. Obshch*, 6:231–298, 1957. In Russian.
- [Adian et al., 1980] S. I. Adian, W. W. Boone, and G. Higman, editors. *Word Problems II*. North-Holland, Amsterdam, 1980.
- [Arnold and Nivat, 1980a] A. Arnold and M. Nivat. Metric interpretations of infinite trees and semantics of nondeterministic recursive programs. *Theoretical Computer Science*, 11:181–205, 1980.
- [Arnold and Nivat, 1980b] A. Arnold and M. Nivat. The metric space of infinite trees: algebraic and topological properties. *Fundamenta Informaticae*, 4:445–476, 1980.
- [Babbage, 1989] C. Babbage. *Collected Works. Volume III: Analytical Engine*. W. Pickering, London, 1989.
- [Baeten and Weijland, 1990] J. C. M. Baeten and P. Weijland. *Process Algebra*. Cambridge University Press, Cambridge, 1990.
- [Barwise, 1977] J. Barwise, editor. *Handbook of Mathematical Logic*. North-Holland, Amsterdam, 1977.
- [Baumslag and Miller III, 1992] G. Baumslag and C. F. Miller III, editors. *Algorithms and Classification in Combinatorial Group Theory*. MSRI Publications 23. Springer-Verlag, New York, 1992.
- [Baur, 1974] W. Baur. Rekursive algebren mit kettenbedingungen. *Zeitschrift für Mathematisch Logik und Grundlagen der Mathematik*, 20:37–46, 1974.
- [Beeson, 1985] M. J. Beeson. *Foundations of Constructive Mathematics*. Springer-Verlag, Berlin, 1985.
- [Berger, 1993] U. Berger. Total sets and objects in domain theory. *Annals of Pure and Applied Logic*, 60:91–117, 1993.
- [Bergstra and Heering, 1994] J. A. Bergstra and J. Heering. Which data types have  $\omega$ -complete initial algebra specifications? *Theoretical Computer Science*, 124:149–168, 1994.

- [Bergstra and Klop, 1983] J. A. Bergstra and J. W. Klop. Initial algebra specifications for parameterized data types. *Elektronische Informationsverarbeitung und Kybernetik*, 19: 7–31, 1983.
- [Bergstra and Tucker, 1979] J. A. Bergstra and J. V. Tucker. Algebraic specifications of computable and semicomputable data structures. Technical Report IW 115, Mathematical Centre, Department of Computer Science, Amsterdam, 1979.
- [Bergstra and Tucker, 1980a] J. A. Bergstra and J. V. Tucker. A characterisation of computable data types by means of a finite equational specification method. In J. W. de Bakker and J. van Leeuwen, editors, *Automata, languages and programming (ICALP), Seventh Colloquium, Noordwijkerhout, 1980*, pages 76–90. Lecture Notes in Computer Science 81, Springer-Verlag, Berlin, 1980.
- [Bergstra and Tucker, 1980b] J. A. Bergstra and J. V. Tucker. A natural data type with a finite equational final semantics specification, but no effective equational initial specification. *Bulletin of the EATCS*, 11:23–33, 1980.
- [Bergstra and Tucker, 1981] J. A. Bergstra and J. V. Tucker. Algebraically specified programming systems and Hoare's logic. In S. Even and O. Kariv, editors, *Automata, Languages and Programming (ICALP), Eighth Colloquium, Acre, 1981*, pages 348–362. Lecture Notes in Computer Science 115, Springer-Verlag, Berlin, 1981.
- [Bergstra and Tucker, 1983a] J. A. Bergstra and J. V. Tucker. The completeness of the algebraic specification methods for computable data types. *Inf. and Control*, 54:186–200, 1983.
- [Bergstra and Tucker, 1983b] J. A. Bergstra and J. V. Tucker. Initial and final algebra semantics for data type specifications: two characterisation theorems. *SIAM Journal on Computing*, 12:366–387, 1983.
- [Bergstra and Tucker, 1987] J. A. Bergstra and J. V. Tucker. Algebraic specifications of computable and semicomputable data types. *Theoretical Computer Science*, 50:137–181, 1987.
- [Bergstra and Tucker, 1990] J. A. Bergstra and J. V. Tucker. The inescapable stack: an exercise in algebraic specification with total functions. Technical Report P8804, University of Amsterdam, 1990.
- [Bergstra and Tucker, 1993a] J. A. Bergstra and J. V. Tucker. The data type variety of stack algebras. In H. Barendregt, M. Bezem, and J. W. Klop, editors, *Dirk van Dalen Festschrift*, pages 9–40. Department of Philosophy, University of Utrecht, 1993.
- [Bergstra and Tucker, 1993b] J. A. Bergstra and J. V. Tucker. Equational specifications for computable data types: six hidden functions suffice and other sufficiency bounds. In K. Meinke and J. V. Tucker, editors, *Many Sorted Logic and its Applications*, pages 89–102. J. Wiley and Sons, Chichester, 1993.

- [Bergstra and Tucker, 1993c] J. A. Bergstra and J. V. Tucker. On bounds for the specification of finite data types by means of equations and conditional equations. In K. Meinke and J. V. Tucker, editors, *Many Sorted Logic and its Applications*, pages 103–122. J. Wiley and Sons, Chichester, 1993.
- [Bergstra and Tucker, 1993d] J. A. Bergstra and J. V. Tucker. Equational specifications, complete term rewriting systems, and computable and semicomputable algebras, University College of Swansea, Department of Computer Science Research Report, 1993.
- [Bergstra et al., 1981] J. A. Bergstra, M. Broy, M. Wirsing, and J. V. Tucker. On the power of algebraic specifications. In J. Gruska and M. Chytil, editors, *Mathematical Foundations of Computer Science 1981, Proceedings Strbske Pleso, Czechoslovakia*, pages 192–204. Lecture Notes in Computer Science 118, Springer-Verlag, Berlin, 1981.
- [Bergstra et al., 1989] J. A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification*. Addison-Wesley, Wokingham, England, 1989.
- [Birkhoff, 1933] G. Birkhoff. On the combination of subalgebras. *Proceedings of the Cambridge Philosophical Society*, 29:441–464, 1933.
- [Birkhoff, 1935] G. Birkhoff. On the structure of abstract algebras. *Proceedings of the Cambridge Philosophical Society*, 31:433–454, 1935.
- [Birkhoff and Lipson, 1970] G. Birkhoff and J. D. Lipson. Heterogeneous algebras. *Journal of Combinatorial Theory*, 8:115–133, 1970.
- [Boone, 1955a] W. W. Boone. Certain simple, unsolvable problems of group theory IV. *Indagationes Mathematicae*, 17:571–577, 1955.
- [Boone, 1955b] W. W. Boone. Certain simple, unsolvable problems of group theory V, VI. *Indagationes Mathematicae*, 19:22–27, 227–232, 1955.
- [Boone et al., 1973] W. W. Boone, F. B. Cannonito, and R. C. Lyndon, editors. *Word Problems*. North-Holland, Amsterdam, 1973.
- [Boone and Higman, 1974] W. W. Boone and G. Higman. An algebraic characterization of the unsolvability of the word problem. *Journal of the Australian Mathematical Society*, 18:41–53, 1974.
- [Boone and Rogers, 1966] W. W. Boone and H. Rogers. On a problem of J. H. C. Whitehead and a problem of Alonzo Church. *Mathematica Scandinavica*, 19: 185–192, 1966.
- [Bridges and Richman, 1987] D. Bridges and F. Richman. *Varieties of Constructive Mathematics*, London Mathematical Society Lecture Notes Series 97, Cambridge University Press, Cambridge, 1987.
- [Brouwer, 1975] L. E. J. Brouwer. *Collected Works I*. North-Holland, Amsterdam, 1975.
- [Burris and Sankappanavar, 1981] S. Burris and H. P. Sankappanavar. *A Course in Universal Algebra*. Springer-Verlag, Berlin, 1981.

- [Cannonito and Gatterdam, 1973] F. B. Cannonito and R. W. Gatterdam. The computability of group constructions. Part 1. In W. W. Boone, F. B. Cannonito and R. C. Lyndon, editors, *Word Problems*, pages 365–400, North-Holland, Amsterdam, 1973.
- [Ceitin, 1962] G. S. Ceitin. Algorithmic operators in constructive metric spaces. *Trudy Mat. Inst. Steklov*, 67:295–361, 1962. English translation, *American Mathematical Society Translations*, 64: 1–80, 1967.
- [Chandler and Magnus, 1982] B. Chandler and W. Magnus. *The History of Combinatorial Group Theory: A Case Study in the History of Ideas*. Springer-Verlag, New York, 1982.
- [Chang and Keisler, 1990] C. C. Chang and H. J. Keisler. *Model Theory*. North-Holland, Amsterdam, 1990.
- [Cohn, 1981] P. M. Cohn. *Universal Algebra*, D.Reidel, Dordrecht, 1981.
- [Crossley, 1981] J. N. Crossley. *Aspects of effective algebra. Proceedings of a conference at Monash University, Australia, 1-4 August, 1979*. Upside Down A Book Company, Steel's Creek, Australia, 1981.
- [Cutland, 1980] N. J. Cutland. *Computability: An Introduction To Recursive Function Theory*. Cambridge University Press, Cambridge, 1980.
- [Davis et al., 1976] M. Davis, Y. Matijasevic, and J. Robinson. Hilbert's tenth problem; positive aspects of a negative solution. In F. E. Browder, editor, *Mathematical Developments Arising from Hilbert's Problems*, pages 323–378. American Mathematical Society, Providence, 1976.
- [de Bakker and Rutten, 1992] J. W. de Bakker and J. J. M. M. Rutten, editors. *Ten Years of Concurrency Semantics*. World Scientific, London, 1992.
- [de Bakker and Zucker, 1982] J. W. de Bakker and J. I. Zucker. Processes and the denotational semantics of concurrency. *Information and Control*, 54:70–120, 1982.
- [Dehn, 1910] M. Dehn. Über die Topologie des dreidimensionalen Raumes. *Mathematische Annalen*, 69:137–168, 1910.
- [Dehn, 1911] M. Dehn. Über unendliche diskontinuierliche Gruppen. *Mathematische Annalen*, 71:116–144, 1911.
- [Dehn, 1987] M. Dehn. *Papers on Group Theory and Topology*. Springer-Verlag, New York, 1987.
- [Dershowitz and Jouannaud, 1990] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Methods and Semantics*, pages 243–320, North-Holland, Amsterdam, 1990.
- [Dubbey, 1978] J. M. Dubbey. *The Mathematical Work of Charles Babbage*. Cambridge University Press, Cambridge, 1978.
- [Dugundji, 1966] J. Dugundji. *Topology*. Allyn and Bacon, Boston, 1966.



- [Ebbinghaus, 1987] H.-D. Ebbinghaus, editor.  *$\Omega$ -Bibliography of Mathematical Logic: Model Theory*. Springer-Verlag, Berlin, 1987.
- [Ehrig and Mahr, 1985] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. EATCS Monographs in Theoretical Computer Science 6. Springer-Verlag, Berlin, 1985.
- [Ershov, 1973] Y. L. Ershov. Theorie der Numerierungen I. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 19:289–388, 1973.
- [Ershov, 1974] Y. L. Ershov. Maximal and everywhere defined functionals. *Algebra and Logic*, 13:374–397, 1974.
- [Ershov, 1975] Y. L. Ershov. Theorie der Numerierungen II. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 21:473–584, 1975.
- [Ershov, 1976] Y. L. Ershov. Hereditarily effective operations. *Algebra and Logic*, 15:642–654, 1976.
- [Ershov, 1977a] Y. L. Ershov. The model  $C$  of the partial continuous functionals. In R. O. Gandy and J. M. E. Hyland, editors, *Logic Colloquium 76*, pages 455–467. North-Holland, Amsterdam, 1977.
- [Ershov, 1977b] Y. L. Ershov. Theorie der Numerierungen III. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 23:289–371, 1977.
- [Ershov, 1979] Y. L. Ershov. *Theory of Numberings*, in Russian, 1979.
- [Feferman, 1975] S. Feferman. Impredicativity of the existence of the largest divisible sub-group of an abelian  $p$ -group. In D. H. Saracino and V. B. Weispfenning, editors, *Model Theory and Algebra*. Lecture Notes in Mathematics 498, Springer-Verlag, Berlin, 1975.
- [Fenstad, 1980] J. E. Fenstad. *General Recursion Theory: An Axiomatic Approach*. Springer-Verlag, Berlin, 1980.
- [Fitting, 1981] M. Fitting. *Fundamentals of Generalized Recursion Theory*. North-Holland, Amsterdam, 1981.
- [Fröhlich and Shepherdson, 1956] A. Fröhlich and J. C. Shepherdson. Effective procedures in field theory. *Philosophical Transactions Royal Society London*, (A) 248:407–432, 1956.
- [Gagliardi and Tulipani, 1990] G. Gagliardi and S. Tulipani. On algebraic specifications of computable algebras with the discriminator technique. *Theoretique Informatique et Appliques*, 24:429–440, 1990.
- [Gandy, 1988] R. Gandy. The confluence of ideas in 1936. In R. Herkin, editor, *The Universal Turing Machine*, pages 55–111. Oxford University Press, Oxford, 1988.
- [Goguen et al., 1977] J. A. Goguen, J. Thatcher, E. Wagner, and J. B. Wright. Initial algebra semantics and continuous algebras. *Journal of the Association for Computing Machinery*, 24:68–95, 1977.



- [Goguen *et al.*, 1978] J. A. Goguen, J. W. Thatcher, and E. G. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In R. T. Yeh, editor, *Current trends in programming methodology. IV. Data structuring*, pages 80–149. Prentice-Hall, Engelwood Cliffs, New Jersey, 1978.
- [Goncharov, 1980a] S. Goncharov. Automorphisms of models. *Algebra and Logic*, 19:45–58, 1980.
- [Goncharov, 1980b] S. Goncharov. Autostability of models and abelian groups. *Algebra and Logic*, 19:13–27, 1980.
- [Goncharov, 1981] S. Goncharov. Groups with a finite number of constructions. *Soviet Math. Doklady*, 23:58–61, 1981.
- [Goodstein, 1961] R. L. Goodstein. *Recursive Analysis*. North-Holland, Amsterdam, 1961.
- [Grätzer, 1979] G. Grätzer. *Universal Algebra*. Springer-Verlag, Berlin, 1979.
- [Grzegorzczuk, 1955] A. Grzegorzczuk. Computable functionals. *Fundamenta Mathematicae*, 42:168–202, 1955.
- [Grzegorzczuk, 1957] A. Grzegorzczuk. On the definitions of computable real continuous functions. *Fundamenta Mathematicae*, 44:61–71, 1957.
- [Guessarian, 1981] I. Guessarian. *Algebraic Semantics*. Lecture Notes in Computer Science 99, Springer-Verlag, Berlin, 1981.
- [Guttag, 1975] J. V. Guttag. *The specification and application to programming of abstract data types*. PhD thesis, University of Toronto, 1975.
- [Hennessy, 1988] M. Hennessy. *Algebraic Theory of Processes*. MIT Press, 1988.
- [Hermann, 1926] G. Hermann. Die frage der endlichen vielen schritte in der theorie der polynomideale. *Mathematische Annalen*, 95:736–788, 1926.
- [Heyting, 1959] A. Heyting, editor. *Constructivity in Mathematics*. North-Holland, Amsterdam, 1959.
- [Higman, 1961] G. Higman. Subgroups of finitely presented groups. *Proceedings Royal Society London Series A*, 262:455–475, 1961.
- [Hinman, 1987] P. G. Hinman, editor.  *$\Omega$ -Bibliography of Mathematical Logic: Recursion Theory*. Springer-Verlag, Berlin, 1987.
- [Hodges, 1993] W. Hodges. *Model Theory*. Cambridge University Press, Cambridge, 1993.
- [Holden *et al.*, 1992] A. V. Holden, J. V. Tucker, H. Zhang, and M. Poole. Coupled map lattices as computational systems. *American Institute of Physics - Chaos*, 2:367–376, 1992.
- [Kasymov, 1987] N. Kh. Kasymov. Algebras with residually finite positively presented expansions. *Algebra and Logic*, 26:715–730, 1987.
- [Kelley, 1955] J. L. Kelley. *General Topology*. Van Nostrand, Princeton, 1955.

- [Kleene, 1959] S. C. Kleene. Countable functionals. In A. Heyting, editor, *Constructivity in Mathematics*, pages 81–100. North-Holland, Amsterdam, 1959.
- [Kleene, 1981] S. C. Kleene. Origins of recursive function theory. *Annals of the History of Computing*, 3:52–67, 1981.
- [Klop, 1992] J. W. Klop. Term rewriting systems. In S. Abramsky, D. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 2*, pages 1–116. Oxford University Press, Oxford, 1992.
- [Ko, 1991] K. Ko. *Complexity Theory of Real Functions*. Birkhäuser, Basel, 1991.
- [Kosiuczenko and Meinke, 1994] P. Kosiuczenko and K. Meinke. On the power of higher-order algebraic specification methods. Department of Computer Science, University College of Swansea, Report, 1994.
- [Kreisel, 1959] G. Kreisel. Interpretation of analysis by means of constructive functionals of finite types. In A. Heyting, editor, *Constructivity in Mathematics*, pages 101–128. North-Holland, Amsterdam, 1959.
- [Kreisel et al., 1959] G. Kreisel, D. Lacombe, and J. R. Shoenfield. Partial recursive functionals and effective operations. In A. Heyting, editor, *Constructivity in Mathematics*, pages 195–207. North-Holland, Amsterdam, 1959.
- [Kreitz and Weihrauch, 1984] C. Kreitz and K. Weihrauch. A unified approach to constructive and recursive analysis. In E. Börger, W. Oberschelp, M. M. Richter, B. Schinzel, and W. Thomas, editors, *Computation and Proof Theory*, pages 259–278. Lecture Notes in Mathematics 1104, Springer-Verlag, Berlin, 1984.
- [Kristiansen and Normann, 1994] L. Kristiansen and D. Normann. Interpreting higher computations as types with totality. *Arch. for Math. Logic*, 1994.
- [Kristiansen, 1993] L. Kristiansen. *Totality in qualitative domains*. PhD Thesis, University of Oslo, 1993.
- [Kronecker, 1882] L. Kronecker. Grundzüge einer arithmetischen theorie der algebraischen grössen. *Journal für Mathematik*, 92:1–122, 1882.
- [Krull, 1953a] W. Krull. Über Polynomzerlegung mit endlich vielen Schritten I. *Mathematische Zeitschrift*, 59:57–60, 1953.
- [Krull, 1953b] W. Krull. Über Polynomzerlegung mit endlich vielen Schritten II. *Mathematische Zeitschrift*, 59:296–298, 1953.
- [Krull, 1954] W. Krull. Über Polynomzerlegung mit endlich vielen Schritten III. *Mathematische Zeitschrift*, 60:109–111, 1954.
- [Kutzler and Lichtenberger, 1983] B. Kutzler and F. Lichtenberger. *Bibliography on Abstract Data Types*. Informatik Fachberichte. Springer-Verlag, Berlin, 1983.

- [Kuznetsov, 1958] A. V. Kuznetsov. Algorithms as operations in algebraic systems. *Uspekhi Mat Nauk*, 13:240–241, 1958. In Russian.
- [Lachlan and Madison, 1970] A. H. Lachlan and E. W. Madison. Computable fields and arithmetically definable ordered fields. *Proceedings of the American Mathematical Society*, 24:803–807, 1970.
- [Lacombe, 1955] D. Lacombe. Extension de la notion de fonction récursive aux fonctions d'une ou plusieurs variables réelles. I, II, III. *Comptes Rendus*, 240, 241:2478–2480, 13–14, 151–153, 1955.
- [Lallement, 1979] G. Lallement. *Semigroups and Combinatorial Applications*. J. Wiley, Chichester, 1979.
- [Leech, 1970] J. Leech. *Computational Problems in Abstract Algebra*. Pergamon Press, Oxford, 1970.
- [Lin, 1981a] C. Lin. The effective content of Ulm's theorem. In J. N. Crossley, editor, *Aspects of Effective Algebra, (Proceedings of a conference at Monash University, Australia, 1–4 August, 1979)*, pages 147–160. Upside Down A Book Company, Steel's Creek, Australia, 1981.
- [Lin, 1981b] C. Lin. Recursively presented Abelian groups: effective  $p$ -group theory I. *Journal of Symbolic Logic*, 46: 617–624, 1981.
- [Lyndon and Schupp, 1977] R. C. Lyndon and P. E. Schupp. *Combinatorial Group Theory*. Springer-Verlag, Berlin, 1977.
- [MacIntyre, 1972] A. MacIntyre. On algebraically closed groups. *Annals of Mathematics*, 96:53–97, 1972.
- [Madison, 1970] E. W. Madison. A note on computable real fields. *Journal of Symbolic Logic*, 35:239–241, 1970.
- [Magnus et al., 1976] W. Magnus, A. Karass, and D. Solitar. *Combinatorial Group Theory*. Dover, New York, 1976.
- [Mal'cev, 1961] A. I. Mal'cev. Constructive algebras I. *Russian Mathematical Surveys*, 16:77–129, 1961. Also in B. F. Wells III, editor, *The Metamathematics of Algebraic Systems. Collected Papers: 1936–1967*, pages 148–212. North-Holland, Amsterdam, 1971.
- [Mal'cev, 1970] A. I. Mal'cev. *Algorithms and Recursive Functions*. Wolters-Noordhoff, Groningen, 1970.
- [Mal'cev, 1971] A. I. Mal'cev. *The Metamathematics of Algebraic Systems: Collected Papers 1936–1967*. Translated and edited by B. F. Wells III, North-Holland, Amsterdam, 1971.
- [Mal'cev, 1973] A. I. Mal'cev. *Algebraic Systems*, Springer-Verlag, Berlin, 1973.
- [Markov, 1947] A. A. Markov. On the impossibility of certain algorithms in the theory of associative systems. *Doklady Akad Sci USSR*, 55:583–586, 1947. In Russian.
- [Marongiu and Tulipani, 1987] G. Marongiu and S. Tulipani. Finite algebraic specifications of semicomputable data types. In H. Ehrig et al.,

- editor, *TAPSOFT 87. Vol I*, pages 111–122. Lecture Notes in Computer Science 249, Springer-Verlag, Berlin, 1987.
- [Marongiu and Tulipani, 1989] G. Marongiu and S. Tulipani. On a conjecture of Bergstra and Tucker. *Theoretical Computer Science*, 67:87–97, 1989.
- [Matiyasevich, 1993] Y. Matiyasevich. *Hilbert's Tenth Problem*. MIT Press, Cambridge, Massachusetts, 1993.
- [McKenzie and Valeriote, 1989] R. McKenzie and M. Valeriote. *The Structure of Decidable Locally Finite Varieties*. Birkhäuser, Basel, 1989.
- [McKenzie et al., 1987] R. N. McKenzie, G. F. McNulty, and W. F. Taylor. *Algebras, Lattices, Varieties. Volume I*. Wadsworth and Brooke/Cole, Monterey, 1987.
- [McKinsey, 1943] J. C. C. McKinsey. The decision problem for some classes of sentences without quantifiers. *Journal of Symbolic Logic*, 8:61–76, 1943.
- [Meinke, 1992] K. Meinke. Universal algebra in higher types. *Theoretical Computer Science*, 100: 385–417, 1992.
- [Meinke, 1994] K. Meinke. A recursive second order initial algebra specification of primitive recursion. *Acta Informatica*, 31: 329–340, 1994.
- [Meinke and Tucker, 1992] K. Meinke and J. V. Tucker. Universal algebra. In S. Abramsky, D. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 1*, pages 189–411. Oxford University Press, Oxford, 1992.
- [Meseguer and Goguen, 1985] J. Meseguer and J. A. Goguen. Initiality, induction and computability. In M. Nivat and J. Reynolds, editors, *Algebraic Methods in Semantics*, pages 459–541. Cambridge University Press, Cambridge, 1985.
- [Meseguer et al., 1992] J. Meseguer, L. Moss and J. A. Goguen. Final algebras, cosemicomputable algebras, and degrees of unsolvability. *Theoretical Computer Science*, 100:267–302, 1992.
- [Metakides and Nerode, 1977] G. Metakides and A. Nerode. Recursively enumerable vector spaces. *Annals of Mathematical Logic*, 11:147–171, 1977.
- [Metakides and Nerode, 1979] G. Metakides and A. Nerode. Effective content of field theory. *Annals of Mathematical Logic*, 17:289–320, 1979.
- [Miller III, 1971] C. F. Miller III. *On Group-Theoretic Decision Problems and their Classification*. Princeton University Press, 1971.
- [Miller III, 1992] C. F. Miller III. Decision problems for groups. Surveys and reflections. In G. Baumslag and C. F. Miller III, editors, *Algorithms and Classification in Combinatorial Group Theory*. MSRI Publications 23, Springer-Verlag, 1992.



- [Moldestad *et al.*, 1980a] J. Moldestad, V. Stoltenberg-Hansen, and J. V. Tucker. Finite algorithmic procedures and computation theories. *Mathematica Scandinavica*, 46:77–94, 1980.
- [Moldestad *et al.*, 1980b] J. Moldestad, V. Stoltenberg-Hansen, and J. V. Tucker. Finite algorithmic procedures and inductive definability. *Mathematica Scandinavica*, 46:62–76, 1980.
- [Moschovakis, 1964] Y. N. Moschovakis. Recursive metric spaces. *Fundamenta Mathematicae*, 55:215–238, 1964.
- [Moschovakis, 1966] Y. N. Moschovakis. Notation systems and recursive ordered fields. *Composito Mathematica*, 17: 40–71, 1966.
- [Mostowski, 1955] A. Mostowski. Examples of sets definable by means of two and three quantifiers. *Fundamenta Mathematicae*, 42:259–270, 1955.
- [Mostowski, 1957] A. Mostowski. On computable sequences. *Fundamenta Mathematicae*, 44:37–51, 1957.
- [Neumann, 1937] B. H. Neumann. Some remarks on infinite groups. *Journal of the London Mathematical Society*, 12: 120–127, 1937.
- [Neumann, 1973] B. H. Neumann. The isomorphism problem for algebraically closed groups. In W. W. Boone, F. B. Cannonito, and R. C. Lyndon, editors, *Word Problems*, pages 553–562. North-Holland, Amsterdam, 1973.
- [Nivat, 1975] M. Nivat. On the interpretation of polyadic recursive program schemes. *Symposia Mathematica*, 15:255–281, 1975.
- [Normann, 1980] D. Normann. *Recursion on the Countable Functionals*. Lecture Notes in Mathematics 811. Springer-Verlag, Berlin, 1980.
- [Normann, 1989] D. Normann. Kleene-spaces. In R. Ferro *et al.*, editor, *Logic Colloquium '88*, pages 91–109. North-Holland, Amsterdam, 1989.
- [Normann, 1990] D. Normann. Formalizing the notion of total information. In P. P. Petkov, editor, *Mathematical Logic*, pages 67–94. Plenum Press, 1990.
- [Novikov, 1955] P. S. Novikov. On the algorithmic unsolvability of the word problem for group theory. *Trudy Mat. Inst. Steklov*, 44, 1955. In Russian.
- [Novy, 1973] L. Novy. *Origins of Modern Algebra*. Noordhoff, Leiden, 1973.
- [Peacock, 1830] G. Peacock. *A Treatise on Algebra*. Cambridge, 1830.
- [Phillips, 1992] I. C. C. Phillips. Recursion theory. In S. Abramsky, D. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 1*, pages 79–187. Oxford University Press, Oxford, 1992.
- [Plotkin, 1981] G. D. Plotkin. Post-graduate Lecture Notes in Advanced Domain Theory (incorporating the ‘Pisa Notes’). Department of Computer Science, University of Edinburgh, 1981.



- [Post, 1947] E. L. Post. Recursive unsolvability of a problem of Thue. *Journal of Symbolic Logic*, 12:1–11, 1947.
- [Pour-El and Richards, 1989] M. B. Pour-El and J. I. Richards. *Computability in Analysis and Physics*. Springer-Verlag, Berlin, 1989.
- [Rabin, 1958] M. O. Rabin. Recursive unsolvability of group theoretic problems. *Annals of Mathematics*, 67:172–194, 1958.
- [Rabin, 1960] M. O. Rabin. Computable algebra, general theory and theory of computable fields. *Transactions of the American Mathematical Society*, 95:341–360, 1960.
- [Reid, 1970] C. Reid. *Hilbert*. Springer-Verlag, Berlin, 1970.
- [Rice, 1954] H. G. Rice. Recursive real numbers. *Proceedings of the American Mathematical Society*, 5:784–791, 1954.
- [Rodenburg, 1991] P. H. Rodenburg. Algebraic specifiability of data types with minimal computable parameters. *Theoretical Computer Science*, 85:97–116, 1991.
- [Rogers, 1967] H. Rogers. *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, New York, 1967.
- [Rotman, 1994] J. Rotman. *An Introduction to the Theory of Groups*. Allyn and Bacon, Boston, third edition, 1994.
- [Sacks, 1990] G. E. Sacks. *Higher Recursion Theory*. Springer-Verlag, Berlin, 1990.
- [Scott, 1972] D. S. Scott. Continuous lattices. In F. W. Lawvere, editor, *Toposes, Algebraic Geometry and Logic*, pages 97–136. Lecture Notes in Mathematics 274, Springer-Verlag, Berlin, 1972.
- [Scott, 1976] D. S. Scott. Data types as lattices. *SIAM Journal on Computing*, 5:522–587, 1976.
- [Scott, 1982a] D. S. Scott. Domains for denotational semantics. In M. Nielsen and E. M. Schmidt, editors, *Automata, Languages and Programming (ICALP), Ninth Colloquium, Aarhus, 1982*, pages 577–613. Lecture Notes in Computer Science 140, Springer-Verlag, Berlin, 1982.
- [Scott, 1982b] D. S. Scott. Lecture notes on a mathematical theory of computation. In M. Broy and G. Schmidt, editors, *Theoretical Foundations of Programming Methodology*, pages 145–292. Reidel, Dordrecht, 1982.
- [Seidenberg, 1971] A. Seidenberg. On the length of a Hilbert ascending chain. *Proceedings of the American Mathematical Society*, 29:443–450, 1971.
- [Seidenberg, 1974a] A. Seidenberg. Constructions in algebra. *Transactions of the American Mathematical Society*, 197:273–313, 1974.
- [Seidenberg, 1974b] A. Seidenberg. What is noetherian? *Rend. Sem. Mat. Fis. Milano*, 44:55–61, 1974.
- [Shepherdson, 1976] J. C. Shepherdson. On the definition of computable function of a real variable. *Zeitschrift für Mathematische Logik und*

*Grundlagen der Mathematik*, 22:391–402, 1976.

- [Shepherdson, 1985] J. C. Shepherdson. Algorithmic procedures, generalised Turing algorithms, and elementary recursion theory. In L. Harrington *et al.*, editors, *Harvey Friedman's Research on the Foundations of Mathematics*, pages 285–308. North-Holland, Amsterdam, 1985.
- [Shore, 1978] R. Shore. Controlling the dependence degree of a recursively enumerable vector space. *Journal of Symbolic Logic*, 43:13–22, 1978.
- [Smith, 1981a] R. L. Smith. Effective valuation theory. In J. N. Crossley, editor, *Aspects of Effective Algebra, Proceedings of a Conference at Monash University, Australia, 1–4 August, 1979*, pages 232–245. Upside Down A Book Company, Steel's Creek, Australia, 1981.
- [Smith, 1981b] R. L. Smith. Two theorems on autostability in  $p$ -groups. In M. Lerman, J. H. Schmerl, and R. I. Soare, editors, *Logic Year 1979–80*, pages 302–311. Lecture Notes in Mathematics 859, Springer-Verlag, Berlin, 1981.
- [Smyth, 1988] M. B. Smyth. Quasi uniformities: reconciling domains with metric spaces. In A. M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Mathematical Foundations of Programming Language Semantics*, pages 236–253. Lecture Notes in Computer Science 298, Springer-Verlag, Berlin, 1988.
- [Smyth, 1992] M. B. Smyth. Topology. In S. Abramsky, D. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 1*, pages 641–761. Oxford University Press, Oxford, 1992.
- [Smyth and Plotkin, 1982] M. B. Smyth and G. D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal of Computing*, 11:761–783, 1982.
- [Spreen, 1990] D. Spreen. A characterization of effective topological spaces. In K. Ambos-Spies *et al.*, editor, *Recursion Theory Week, Proceedings, Oberwolfach, 1989*, pages 363–388. Lecture Notes in Mathematics 1432, Springer-Verlag, 1990.
- [Spreen, 1991] D. Spreen. A characterization of effective topological spaces II. In G. M. Reed *et al.*, editor, *Topology and Category Theory in Computer Science*, pages 231–255. Oxford University Press, Oxford, 1991.
- [Spreen and Young, 1984] D. Spreen and P. Young. Effective operators in a topological setting. In M. M. Richter *et al.*, editor, *Computation and Proof Theory: Proceedings of the Logic Colloquium '83*, pages 437–451. Lecture Notes in Mathematics 1104, Springer-Verlag, Berlin, 1984.
- [Steinitz, 1910] E. Steinitz. Algebraische Theorie der Körper. *Journal für Mathematik*, 137:167–309, 1910.
- [Stillwell, 1982] J. Stillwell. The word problem and the isomorphism problem for groups. *Bulletin of the American Mathematical Society*, 6:33–56, 1982.

- [Stoltenberg-Hansen and Tucker, 1980] V. Stoltenberg-Hansen and J. V. Tucker. Computing roots of unity in fields. *Bulletin of the London Mathematical Society*, 12:463–471, 1980.
- [Stoltenberg-Hansen and Tucker, 1988] V. Stoltenberg-Hansen and J. V. Tucker. Complete local rings as domains. *Journal of Symbolic Logic*, 53:603–624, 1988.
- [Stoltenberg-Hansen and Tucker, 1991] V. Stoltenberg-Hansen and J. V. Tucker. Algebraic and fixed point equations over inverse limits of algebras. *Theoretical Computer Science*, 87:1–24, 1991.
- [Stoltenberg-Hansen and Tucker, 1993] V. Stoltenberg-Hansen and J. V. Tucker. Infinite systems of equations over inverse limits and infinite synchronous concurrent algorithms. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Semantics: Foundations and Applications*, pages 531–562. Lecture Notes in Computer Science 666, Springer-Verlag, Berlin, 1993.
- [Stoltenberg-Hansen and Tucker, 1995] V. Stoltenberg-Hansen and J. V. Tucker. *Computable Rings and Fields: An Introduction*. Cambridge University Press, Cambridge, to appear, 1995.
- [Stoltenberg-Hansen et al., 1994] V. Stoltenberg-Hansen, I. Lindström, and E. R. Griffor. *Mathematical Theory of Domains*. Cambridge Tracts in Theoretical Computer Science, Vol 22. Cambridge University Press, Cambridge, 1994.
- [Sturm, 1835] C.-F. Sturm. Mémoire sur la résolution des équations numériques. *Annales de Mathématiques Pures et Appliquées*, 6:271–318, 1835.
- [Taylor, 1979] W. Taylor. Equational logic. *Houston Journal of Mathematics*, pages 1–83, 1979.
- [Thue, 1914] A. Thue. Probleme über Veränderungen von Zeichenreihen nach gegebenen Regeln. *Skr. Vid. Kristiania, I Mat. Naturv. Klasse*, 10:34, 1914.
- [Thue, 1977] A. Thue. *Axel Thue's Selected Works*. University of Oslo Press, Oslo, 1977.
- [Troelstra, 1988] A. S. Troelstra. On the early history of intuitionistic logic. Technical Report ML-88-04, University of Amsterdam, 1988.
- [Troelstra, 1991] A. S. Troelstra. History of constructivism in the twentieth century. Technical Report ML-91-05, University of Amsterdam, 1991.
- [Tucker, 1977] J. V. Tucker. *Computability as an algebraic property*. PhD Thesis, School of Mathematics, University of Bristol, 1977.
- [Tucker, 1980a] J. V. Tucker. Computability and the algebra of fields: Some affine constructions. *Journal of Symbolic Logic*, 45:103–120, 1980.
- [Tucker, 1980b] J. V. Tucker. Computing in algebraic systems. In F. R. Drake and S. S. Wainer, editors, *Recursion Theory, its Generalisations*

- and Applications, pages 215–235. London Mathematical Society Lecture Note Series 45, Cambridge University Press, Cambridge, 1980.
- [Tucker, 1991] J. V. Tucker. Theory of computation and specification over abstract data types and its applications. In F. L. Bauer, editor, *Logic, Algebra and Computation, Proceedings of NATO Summer School 1989 at Maktoberdorf*, pages 1–39. Springer-Verlag, Berlin, 1991.
- [Tucker and Zucker, 1988] J. V. Tucker and J. I. Zucker. *Program Correctness over Abstract Data Types with Error-State Semantics*. North-Holland, Amsterdam, 1988.
- [Tucker and Zucker, 1991] J. V. Tucker and J. I. Zucker. Examples of semi-computable sets of real and complex numbers. In J. P. Myers Jr and M. J. O'Donnell, editors, *Constructivity in Computer Science*, pages 179–198. Lecture Notes in Computer Science 613, Springer-Verlag, Berlin, 1991.
- [Tucker and Zucker, 1992] J. V. Tucker and J. I. Zucker. Theory of computation over stream algebras, and its applications. In I. M. Havel and V. Koubek, editors, *Mathematical Foundations of Computer Science 1992, 17th International Symposium, Prague*, pages 62–80. Lecture Notes in Computer Science 629, Springer-Verlag, Berlin, 1992.
- [Tucker and Zucker, 1994] J. V. Tucker and J. I. Zucker. Computable functions on stream algebras. In H. Schwichtenberg, editor, *Proof and Computation*, pages 341–283, Springer-Verlag, Berlin, 1994.
- [Turing, 1936] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society, Ser.2*, 42:230–265, 1936.
- [Turing, 1950] A. Turing. The word problem in semigroups with cancellation. *Annals of Mathematics*, 52: 491–505, 1950.
- [van der Waerden, 1930a] B. L. van der Waerden. Eine bemerkung über die unzerlegbarkeit von polynomen. *Mathematische Annalen*, 102:738–739, 1930.
- [van der Waerden, 1930b] B. L. van der Waerden. *Moderne Algebra*. Julius Springer, Berlin, first edition, 1930.
- [van der Waerden, 1970] B. L. van der Waerden. *Algebra, Volumes 1–2*. Ungar, New York, 1970.
- [van Wijngaarden, 1966] A. van Wijngaarden. Numerical analysis as an independent science. *BIT*, 6:68–81, 1966.
- [Vandiver, 1936] H. S. Vandiver. On the ordering of real algebraic numbers by constructive methods. *Annals of Mathematics*, 37:7–16, 1936.
- [Vrancken, 1987] J. L. M. Vrancken. The algebraic specification of semi-computable data types. In D. Sanella and A. Tarlecki, editors, *Recent Trends in Data Type Specification*, Lecture Notes in Computer Science 332, pages 249–259, Springer-Verlag, Berlin, 1987.
- [Wechler, 1991] W. Wechler. *Universal Algebra for Computer Scientists*.



- EATCS Monographs on Theoretical Computer Science 25. Springer-Verlag, Berlin, 1991.
- [Weihrauch, 1987] K. Weihrauch. *Computability*. EATCS Monographs on Theoretical Computer Science 9. Springer-Verlag, Berlin, 1987.
- [Wirsing, 1990] M. Wirsing. Algebraic specifications. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science. Volume B: Formal Models and Semantics*, pages 675–788. North-Holland, Amsterdam, 1990.
- [Wüssing, 1984] H. Wüssing. *The Genesis of the Abstract Group Concept*. MIT Press, Cambridge, MA, 1984.
- [Zassenhaus, 1970] H. Zassenhaus. A real root calculus. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 383–392. Pergamon Press, Oxford, 1970.



# Abstract interpretation: a semantics-based tool for program analysis

Neil D. Jones and Flemming Nielson

---

## Contents

1	Introduction . . . . .	528
1.1	Goals and motivations . . . . .	528
1.2	Relation to program verification and transformation . . . . .	535
1.3	The origins of abstract interpretation . . . . .	535
1.4	A sampling of data-flow analyses . . . . .	536
1.5	Outline . . . . .	538
2	Basic concepts and problems to be solved . . . . .	539
2.1	A naive analysis of the simple program . . . . .	540
2.2	Accumulating semantics for imperative programs . . . . .	542
2.3	Correctness and safety . . . . .	548
2.4	Scott domains, lattice duality, and meet versus join . . . . .	555
2.5	Abstract values viewed as relations or predicates . . . . .	556
2.6	Important points from earlier sections . . . . .	560
2.7	Towards generalizing the Cousot framework . . . . .	561
2.8	Proving safety by logical relations . . . . .	565
3	Abstract interpretation using a two-level metalanguage . . . . .	568
3.1	Syntax of metalanguage . . . . .	569
3.2	Specification of analyses . . . . .	575
3.3	Correctness of analyses . . . . .	588
3.4	Induced analyses . . . . .	595
3.5	Expected forms of analyses . . . . .	604
3.6	Extensions and limitations . . . . .	609
4	Other analyses, language properties, and language types . . . . .	610
4.1	Approaches to abstract interpretation . . . . .	611
4.2	Examples of instrumented semantics . . . . .	614
4.3	Analysis of functional languages . . . . .	616
4.4	Complex abstract values . . . . .	621

4.5	Abstract interpretation of logic programs . . . . .	623
5	Glossary . . . . .	627

## 1 Introduction

Desirable mathematical background for this chapter includes

- basic concepts such as lattices, complete partial orders, homomorphisms, etc.
- the elements of domain theory, e.g. as in the chapter by Abramsky in volume 3 of this Handbook, or the books [Schmidt, 1986] or [Nielson and Nielson, 1992a].
- the elements of denotational semantics, e.g. as in the chapter by Tennent in volume 3 of this Handbook, or the books [Schmidt, 1986] or [Nielson and Nielson, 1992a].
- interpretations as used in logic.

There will be some use of structural operational semantics [Kahn, 1987], [Plotkin, 1981], [Nielson and Nielson, 1992a], for example deduction rules for a program's semantics and type system. The use of category theory will be kept to a minimum but would be a useful background for the domain-related parts of section 3.

### 1.1 Goals and motivations

Our primary goal is to obtain as much information as possible about a program's possible run-time behaviour without actually having to run it on all input data; and to do this automatically. A widely used technique for such program analysis is nonstandard execution, which amounts to performing the program's computations using *value descriptions* or *abstract values* in place of the actual computed values. The results of the analysis must describe *all possible program executions*, in contrast to profiling and other run-time instrumentation which describe only one run at a time. We use the term "abstract interpretation" for a semantics-based version of nonstandard execution. Abramsky [1987] contains an overview and a collection of articles on the subject.

Nonstandard execution can be roughly described as follows:

- perform commands (or evaluate expressions, satisfy goals etc.) using stores, values, . . . drawn from abstract value domains instead of the actual stores, values, . . . used in computations
- deduce information about the program's computations on actual input data from the resulting abstract descriptions of stores, values, . . .

One reason for using abstract stores, values, . . . instead of the actual ones is for computability: to ensure that analysis results are obtained in finite

time. Another is to obtain results that describe the result of computations on a set of possible inputs. The “rule of signs” is a simple, familiar abstract interpretation using abstract values “positive”, “negative”, and “?” (the latter is needed to express, for example, the result of adding a positive and a negative number).

Another classical example is to check arithmetic computations by “casting out nines”, a method using abstract values 0, 1, ..., 8 to detect errors in hand computations. The idea is to perform a series of additions, subtractions, and multiplications with the following twist: whenever a result exceeds 9, it is replaced by the sum of its digits (repeatedly if necessary). The result obtained this way should equal the sum modulo 9 of the digits of the result obtained by the standard arithmetic operations. For example, consider the alleged calculation

$$123 * 457 + 76543 =? = 132654$$

This is checked by reducing 123 to 6, 457 to 7, and 76543 to 7, and then reducing  $6 * 7$  to 42 and so further to 6, and finally  $6 + 7$  is reduced to 4. This differs from 3, the sum modulo 9 of the digits of 132654, so the calculation was incorrect. That the method is correct follows from:

$$\begin{aligned}(10a \pm b) \bmod 9 &= (a \pm b) \bmod 9 \\ a * b \bmod 9 &= (a \bmod 9 * b \bmod 9) \bmod 9 \\ a + b \bmod 9 &= (a \bmod 9 + b \bmod 9) \bmod 9\end{aligned}$$

The method abstracts the actual computation by only recording values modulo 9. Even though much information is lost, useful results are still obtained since this implication holds: if the alleged answer modulo 9 differs from the answer got by casting out nines, there is definitely an error.

*On the need for approximation* Due to the unsolvability of the halting problem (and nearly any other question concerning program behaviour), no analysis that always terminates can be exact. Therefore we have only three alternatives:

- Consider systems with a finite number of finite behaviours (e.g. programs without loops) or decidable properties (e.g. type checking as in Pascal). Unfortunately, many interesting problems are not so expressible.
- Ask interactively for help in case of doubt. But experience has shown that users are often unable to infer useful conclusions from the myriads of esoteric facts provided by a machine. This is one reason why interactive program proving systems have turned out to be less useful in practice than hoped.
- Accept *approximate* but correct information.

Consequently most research in abstract interpretation has been concerned with effectively finding “safe” descriptions of program behaviour, yielding answers which, though sometimes too conservative in relation to the program’s actual behaviour, never yield unreliable information. In a formal sense we seek a  $\sqsubseteq$  relation instead of equality. The effect is that the price paid for exact computability is loss of precision.

A natural analogy: abstract interpretation is to formal semantics as numerical analysis is to mathematical analysis. Problems with no known analytic solution can be solved numerically, giving approximate solutions, for example a numerical result  $r$  and an error estimate  $\epsilon$ . Such a result is *reliable* if it is certain that the correct result lies within the interval  $[r - \epsilon, r + \epsilon]$ . The solution is acceptable in practice if  $\epsilon$  is small enough. In general more precision can be obtained at greater computational cost.

*Safety* Abstract interpretation usually deals with discrete non-numerical objects that require a different idea of approximation than the numerical analyst’s. By analogy, the results produced by abstract interpretation of programs should be considered as correct by a pure semantician, as long as the answers are “safe” in the following sense. A boolean question can be answered “true”, “false”, or “I don’t know”, while answers for the rule of signs could be “positive”, “negative”, or “?”. This apparently crude approach is analogous to the numerical analyst’s, and for practical usage the problem is not to give uninformative answers too often, analogous to the problem of obtaining a small  $\epsilon$ .

An approximate program analysis is *safe* if the results it gives can always be depended on. The results are allowed to be imprecise as long as they always err “on the safe side”, so if boolean variable  $J$  is sometimes true, we allow it to be described as “I don’t know”, but not as “false”. Again, in general more precision can be obtained at greater computational cost.

Defining the term “safe” is, however, a bit more subtle than it appears. In applications, e.g. code optimization in a compiler, it usually means “the result of abstract interpretation may safely be used for program transformation”, i.e. without changing the program’s semantics. To define safety it is essential to understand precisely how the abstract values are to be interpreted in relation to actual computations.

For an example suppose we have a function definition

$$f(X_1, \dots, X_n) = \text{exp}$$

where *exp* is an expression in  $X_1, \dots, X_n$ . Two subtly different *dependency analyses* associate with *exp* a subset of *fs* arguments:

*Analysis I.*

$$\{X_{i_1}, \dots, X_{i_m}\} = \{X_j \mid \text{exp's value depends on } X_j \text{ in at least one computation of } f(X_1, \dots, X_n)\}$$



*Analysis II.*

$$\{X_{i1}, \dots, X_{im}\} = \{X_j \mid \text{exp's value depends on } X_j \text{ in every computation of } f(X_1, \dots, X_n)\}$$

For the example

$$f(W, X, Y, Z) = \text{if } W \text{ then } (X + Y) \text{ else } (X + Z)$$

analysis I yields  $\{W, X, Y, Z\}$ , which is the smallest variable set always sufficient to evaluate the expression. Analysis II yields  $\{W, X\}$ , signifying that regardless of the outcome of the test, evaluation of *exp* requires the values of both *W* and *X*, but not necessarily those of *Y* or *Z*.

These are both dependence analyses but have different modality. Analysis I, for possible dependence, is used in the binding time analysis phase of *partial evaluation*: a program transformation which performs as much as possible of a program's computation, when given knowledge of only some of its inputs. Any variable depending on at least one unknown input in at least one computation might be unknown at specialization time. Thus if any among *W, X, Y, Z* are unknown, then the value of *exp* will be unknown.

Analysis II, for definite dependence, is a *need analysis* identifying that the values of *W* and *X* will always be needed to return the value. Such analyses are used to optimize program execution in lazy languages. The basis is that arguments definitely needed in a function call  $f(e_1, e_2, e_3, e_4, e_5)$  may be pre-evaluated, e.g. using "call by value" for  $e_2$  and  $e_3$ , instead of the more expensive "call by need".

*Strictness.* Finding needed variables involves tracing possible computation paths and variable usages. For mathematical convenience, many researchers work with a slightly weaker notion. A function is defined to be "strict" in variable *A* if whenever *A*'s value is undefined, the value of *exp* will also be undefined, regardless of the other variables' values. Formally this means: if *A* has the undefined value  $\perp$  then *exp* evaluates to  $\perp$ . Clearly *f* both needs and is strict in variables *W* and *X* in the example. For another, *X* is strict in a definition  $f(X) = f(X) + 1$  since  $f(\perp) = \perp$ , even though it is not needed.

*Violations of safety* In practice unsafe data-flow analyses are sometimes used on purpose. For example, highly optimizing compilers may perform "code motion", where code that is invariant in a loop may be moved to a point just before the loop's entry. This yields quite substantial speedups for frequently iterated loops but it can also change termination properties: the moved code will be performed once if placed before the loop, even if the loop exit occurs without executing the body. Thus the transformed program could go into a nonterminating computation not possible before "optimization".

The decision as to whether such efficiency benefits outweigh problems



of semantic differences can only be taken on pragmatic grounds. If one takes a “completely pure view” even using the associative law to rearrange expressions may fail on current computers.

We take a purist’s view in this chapter, insisting on safe analyses and solid semantic foundations, and carefully defining the interpretation of the various abstract values we use.

*Abstract interpretation cannot always be homomorphic* A very well-established way to formulate the faithful simulation of one system by another is by a homomorphism from one algebra to another. Given two (one-sorted) algebras

$$(D, \{a_i : D^{k_i} \rightarrow D\}_{i \in I})$$

and

$$(E, \{b_i : E^{k_i} \rightarrow E\}_{i \in I})$$

with carriers  $D, E$  and operators  $a_i, b_i$ , a *homomorphism* is a function  $\beta : D \rightarrow E$  such that for each  $i$  and  $x_1, \dots, x_{k_i} \in D$

$$\beta(a_i(x_1, \dots, x_{k_i})) = b_i(\beta x_1, \dots, \beta x_{k_i})$$

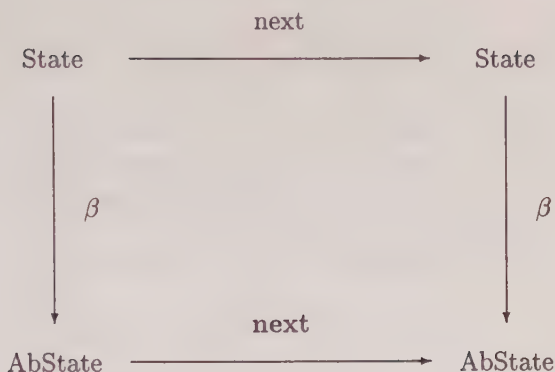
In the examples of sign analysis (to be given later) and casting out nines, abstract interpretation is done by a homomorphic simulation of the operations  $+$ ,  $-$ , and  $*$ . Unfortunately, pure homomorphic simulation is not always sufficient for program analysis.

To examine the problem more closely, consider the example of nonrecursive imperative programs. The “state” of such a program might be a program control point, together with the values of all variables. The semantics is naturally given by defining a *state transition function*, for instance

```
State = Program point × Store
Store = Variable → Value
next  : State → State
```

where we omit formally specifying a language syntax and defining “next” on grounds of familiarity.

Consider the algebra  $(\text{State}, \text{next} : \text{State} \rightarrow \text{State})$  and an abstraction  $(\text{AbState}, \text{next} : \text{AbState} \rightarrow \text{AbState})$ , where  $\text{AbState}$  is a set of abstract descriptions of states. A truly homomorphic simulation of the computation would be a function  $\beta : \text{State} \rightarrow \text{AbState}$  such that the following diagram commutes:



In this case  $\beta$  is a *representation function* mapping real states into their abstract descriptions, and **next** simulates next's effects, but is applied to abstract descriptions.

This elegant view is, alas, not quite adequate for program analysis. For an example, consider sign analysis of a program where

$\text{AbState} = \text{Program point} \times \text{AbStore}$

$\text{AbStore} = \text{Variable} \rightarrow \{+, -, ?\}$

**next** :  $\text{AbState} \rightarrow \text{AbState}$

Representation function  $\beta$  preserves control points and maps each variable into its sign. (The use of abstract value “?”, representing “unknown sign”, will be illustrated later.) If the program contains

$p : Y := X + Y; \text{goto } q$

and the current state is  $(p, [X \mapsto 1, Y \mapsto -2])$  then we have

$$\begin{aligned} \beta(\text{next}((p, [X \mapsto 1, Y \mapsto -2]))) &= \beta((q, [X \mapsto 1, Y \mapsto -1])) \\ &= (q, [X \mapsto +, Y \mapsto -]) \end{aligned}$$

On the other hand the best that **next** can possibly do is:

$$\begin{aligned} \text{next}(\beta((p, [X \mapsto 1, Y \mapsto -2]))) &= \text{next}((p, [X \mapsto +, Y \mapsto -])) \\ &= (q, [X \mapsto +, Y \mapsto ?]) \end{aligned}$$

since  $X + Y$  can be either positive or negative, depending on the exact values of  $X, Y$  (unavailable in the argument of **next**). Thus the desired commutativity fails.

In general the best we can hope for is a *semihomomorphic* simulation. A simple way is to equip  $E$  with a partial order  $\sqsubseteq$ , where  $x \sqsubseteq y$  intuitively means “ $x$  is a more precise description than  $y$ ”, e.g.  $+$   $\sqsubseteq$   $?$ .

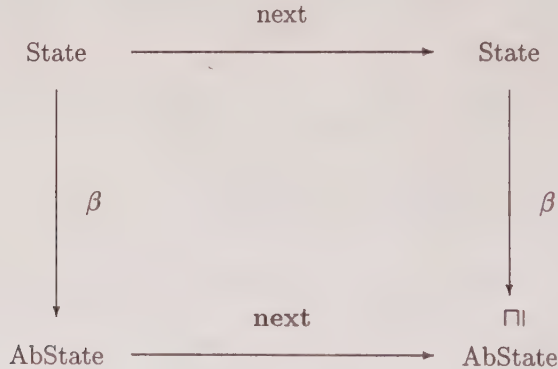
In relation to *safe* value descriptions, as discussed in section 1.1, if  $x$  is a safe description of precise value  $v$ , and  $x \sqsubseteq y$ , then we will also expect  $y$  to be a safe description of  $v$ .

Computations involving abstract values cannot be more precise than those involving actual values, so we weaken the homomorphism restriction by allowing the values computed abstractly to be less precise than the result of exact computation followed by abstraction.

We thus require that for each  $i$  and  $x_1, \dots, x_{k_i} \in D$

$$\beta(a_i(x_1, \dots, x_{k_i})) \sqsubseteq b_i(\beta x_1, \dots, \beta x_{k_i})$$

and that the operations  $b_i$  be monotone. For the imperative language this is described by



The monotonicity condition implies

$$\beta(\text{next}^n(s)) \sqsubseteq \text{next}^n(\beta(s))$$

for all states  $s$  and  $n \geq 0$ , so computations by sequences of state transitions are also safely modelled.

*Abstract interpretation in effect simulates many computations at once* A further complication is that “real world” execution steps cannot be simulated in a one-to-one manner in the “abstract world”. In program fragment

**p: if  $X > Y$  then goto q else goto r**

$\text{next}((p, [X \mapsto +, Y \mapsto +]))$  could yield either  $(q, [X \mapsto +, Y \mapsto +])$  or  $(r, [X \mapsto +, Y \mapsto +])$ , since the approximate descriptions contain too little information to determine the outcome of the test. Operationally this amounts to *nondeterminism*: the argument to **next** does not uniquely determine its result. How is such nondeterminism in the abstract world to be treated?

One way is familiar from finite automata theory: we lift

$$\text{next} : \text{State} \rightarrow \text{State}$$

to work on *sets of states*, namely

$$\wp\text{next} : \wp(\text{State}) \rightarrow \wp(\text{State})$$

defined by

$$\wp\text{next}(\text{state-set}) = \{ \text{next}(s) \mid s \in \text{state-set} \}$$

together with an abstraction function  $\alpha: \wp(\text{State}) \rightarrow \text{AbState}$ . This direction, developed by Cousot and Cousot and described in section 2.2, allows **next** to remain a function.

Another approach is to let  $\beta$  be a relation instead of a function. This approach is described briefly in section 2.8 and is also used in section 3.

The essentially nondeterministic nature of abstract execution implies that abstract interpretation techniques may be used to analyse *nondeterministic programs* as well as deterministic ones. This idea is developed further in [Nielson, 1983].

## 1.2 Relation to program verification and transformation

Program verification has similar goals to abstract interpretation. A major difference is that abstract interpretation emphasizes *approximate* program descriptions obtainable by *fully automatic* algorithms, whereas program verification uses deductive methods which can in principle yield more precise results, but are not guaranteed to terminate. Another difference is that an abstract interpretation, e.g. sign detection, must work uniformly for *all* programs in the language it is designed for. In contrast, traditional program verification requires one to devise a new set of statement invariants for every new program.

Abstract interpretation's major application is to determine the applicability or value of optimization and thus has similar goals to program transformation [Burstall and Darlington, 1977]. However, most program transformation as currently practised still requires considerable human interaction and is so significantly less automatic than abstract interpretation. Further, program transformation often requires proofs that certain transformations can be validly applied; abstract interpretation gives one way to obtain these.

## 1.3 The origins of abstract interpretation

The idea of computing by means of abstract values for analysis purposes is far from new. Peter Naur identified the idea very early and applied it in work on the Gier Algol compiler [Naur, 1965]. He coined the term *pseudo-evaluation* for what was later described as "a process which combines the

operators and operands of the source text in the manner in which an actual evaluation would have to do it, but which operates on descriptions of the operands, not on their values" [Jensen, 1991]. The same basic idea is found in [Reynolds, 1969] and [Sintzoff, 1972]. Sintzoff used it for proving a number of well-formedness aspects of programs in an imperative language, and for verifying termination properties.

These ideas were applied on a larger scale to highly optimizing compilers, often under the names program flow analysis or data-flow analysis [Hecht, 1977], [Aho *et al.*, 1986], [Kam and Ullman, 1977]. They can be used for extraction of more general program properties [Wegbreit, 1975] and have been used for many applications including: generating assertions for program verifiers [Cousot and Cousot, 1977a], program validation [Fosdick and Osterweil, 1976] and [Foster, 1987], testing applicability of program transformations [Nielson, 1985b], compiler generation and partial evaluation [Jones *et al.*, 1989], [Nielson and Nielson, 1988a], estimating program running times [Rosendahl, 1989], and efficiently parallelizing sequential programs [Masdupuy, 1991], [Mercouroff, 1991].

The first papers on automatic program analysis were rather ad hoc, and oriented almost entirely around one application: optimization of target or intermediate code by compilers. Prime importance was placed on efficiency, and the flow analysis algorithms used were not explicitly related to the semantics of the language being analysed. Signs of this can be seen in the well-known unreliability of the early highly optimizing compilers, indicating the need for firmer theoretical foundations.

## 1.4 A sampling of data-flow analyses

We now list some program analyses that have been used for efficient implementation of programming languages. The aim is to show how large the spectrum of interesting program analyses is, and how much they differ from one another. Only a few of these have been given good semantic foundations, so the list could serve as a basis for future work. References include [Aho *et al.*, 1986] and [Muchnick and Jones, 1981].

All concern analysing the subject program's behaviour at particular program points for optimization purposes. The following is a rough classification of the analyses, grouped according to the behavioural properties on which they depend:

**Sets of values, stores, or environments that can occur at a program point**

*Constant propagation* finds out which assignments in a program yield constant values that can be computed at compile time.

*Aliasing analysis* identifies those sets of variables that may refer to the same memory cell.

*Copy propagation* finds those variables whose values equal those of other



variables.

*Destructive updating* recognizes when a new binding of a value to a variable may safely overwrite the variable's previous value, e.g. to reduce the frequency of garbage collection in Lisp [Bloss and Hudak, 1986], [Jensen and Mogensen, 1990], [Mycroft, 1981], [Sestoft, 1988].

*Groundness analysis (in logic programming)* finds out which of a Prolog program's variables can only be instantiated to ground terms [Debray, 1986], [Søndergaard, 1986].

*Sharing analysis (in logic programming)* finds out which variable pairs can be instantiated to terms containing shared subterms [Debray, 1986], [Mellish, 1985], [Søndergaard, 1986].

*Circularity analysis (in logic programming)* finds out which unifications in Prolog can be safely performed without the time-consuming "occur check" [Plaisted, 1984], [Søndergaard, 1986].

### Sequences of variable values

*Variables invariant in loops* identifies those variables in a loop that are assigned the same values every time the loop is executed; used in *code motion*, especially to optimize matrix algorithms.

*Induction variables* identifies loop variables whose values vary regularly each time the loop is executed, also to optimize matrix algorithms.

### Computational past

*Use-definition chains* associates with a reference to  $X$  the set of all assignments  $X := \dots$  that assign values to  $X$  that can "reach" the reference (following the possible flow of program control).

*Available expressions* records the expressions whose values are implicitly available in the values of program variables or registers.

### Computational future

*Live variables* variable  $X$  is *dead* at program point  $p$  if its value will never be needed after control reaches  $p$ , else *live*. Memory or registers holding dead variables may be used for other purposes.

*Definition-use analysis* associates with any assignment  $X := \dots$  the set of all places where the value assigned to  $X$  can be referenced.

*Strictness analysis* given a functional language with normal order semantics, the problem is to discover which parameters in a function call can be evaluated using call by value.

### Miscellaneous

*Mode analysis* to find out which arguments of a Prolog "procedure" are *input*, i.e. will be instantiated when the procedure is entered, and

which are *output*, i.e. will be instantiated as the result of calling the procedure [Mellish, 1985].

*Binding-time analysis* to find out which program variables depend only on a given set of “static program inputs”. Such variables’ values may be precomputed during the optimizing transformation called *partial evaluation* or program specialization [Jones *et al.*, 1993; Jones *et al.*, 1989].

*Interference analysis* to find out which subsets of a program’s commands can be executed so that none in a subset changes variables used by others in the same set. Such sets are candidates for parallel execution on shared memory, vector or data-flow machines.

## 1.5 Outline

Ideally an overview chapter such as this one should describe its area both in breadth and in depth - difficult goals to achieve simultaneously, given the amount of literature and number of different methods used in abstract interpretation. As a compromise section 2 emphasizes overview, breadth, and connections with other research areas, while section 3 gives a more formal mathematical treatment of a domain-based approach to abstract interpretation using a two-level typed lambda calculus. (The motivation is that abstract interpretation of denotational language definitions allows approximation of a wide class of programming language properties.) Section 4 is again an overview, referencing some of the many abstract interpretations that have been seen in the literature. Section 5 contains a glossary briefly describing the many terms that have been introduced. The following is a more detailed overview.

Driven by examples, section 2 introduces several fundamental analysis concepts seen in the literature. The descriptions are informal, few theorems are proved, and some concepts are made more precise later within the framework of section 3.

The section begins with a list of program analyses used by compilers, and does a parity analysis of an example program. The shortcomings of naive analysis methods are pointed out, leading to the need for a more systematic framework. The framework used by Cousot for flow chart programs is introduced, using what we call the “accumulating” semantics.<sup>1</sup>

Appropriate machinery is introduced to approximate the accumulating semantics, and to prove the approximations safe. The distinction between independent attribute and relational analyses is made, and the latter are related to Dijkstra’s predicate transformers. Backwards analyses are then briefly described.

---

<sup>1</sup>There is a terminological problem here: [Cousot and Cousot, 1977b] used the term “static semantics”, but this has other meanings, so several researchers have used the more descriptive “collecting semantics”. Unfortunately this term too has been used in more than one way, so we have invented yet another term: “accumulating semantics”.

It is then shown how domain-based generalizations of these ideas can be applied to languages defined by denotational semantics, thus going far beyond flow chart programs. The main tools used are interpretations and logical relations, and a general technique is introduced for proving safety.

Section 3 uses representation functions and logical relations, rather than abstraction of an accumulating semantics. The approach is *metalanguage* oriented and highly systematic, emphasizing the metalanguage for denotational definitions rather than particular semantic definitions of particular languages. It emphasizes compositionality with respect to domain constructors, and the extension from the approximation of basic values and functions to all the program's domains, analogous to the construction of a free algebra from a set of generators. The components of the following goal are precisely formulated:

abstract interpretation	=	correctness
	+	most precise analyses
	+	implementable analyses

Section 4 illustrates the need to interpret programs over domains other than abstractions of the accumulating semantics. Some program analyses not naturally expressed by abstracting either an accumulating or an *instrumented* semantics are exemplified, showing the need for more sophisticated analysis techniques, and an overview is given of some alternative approaches including tree grammars.

The idea of an “instrumented” semantics is introduced and correctness is discussed. This section is problem oriented, with simulation techniques chosen ad hoc to fit the analysis problem and the language being analysed. It thus centres more around programs’ operational behaviour than the structure of their domains, with particular attention to describing the set of program states reachable in computations on given input data, and to finite description of the set of all computations on given input. The section ends by describing approaches to abstract interpretation of Prolog.

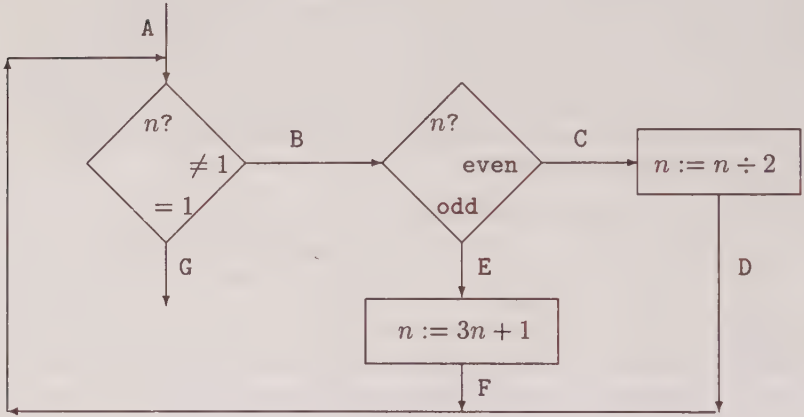
## 2 Basic concepts and problems to be solved

We begin with parity analysis of a very simple example program, and introduce basic concepts only as required. We discuss imperative programs without procedure calls since this familiar program class has a simple semantics and is most often treated in the analysis algorithms found in compiling textbooks. Later sections will discuss functional and logic programs, but many of their analysis problems are also visible, and usually in simpler form, in the imperative context. Throughout this section the reader is encouraged to ask “what is the analogue of this concept in a functional or logic programming framework?”

An example program, where  $\div$  stands for integer division:

```

A: while  $n \neq 1$  do
  B: if  $n$  even
    then (C:  $n := n \div 2$ ; D: )
    else (E:  $n := 3 * n + 1$ ; F: )
  fi
od
G:
  
```



Side remark: Collatz' problem in number theory amounts to determining whether this program terminates for all positive initial  $n$ . To our knowledge it is still unsolved.

## 2.1 A naive analysis of the simple program

*Abstraction of a single execution* If this program is run with initial value  $n = 5$ , then  $n$  takes on values 5, 16, 8, 4, 2 at point B, values 16, 8, 4, 2 at C, etc. Using T to represent "either even or odd" the results of this single run can be abstracted as:

$n$ at A	$n$ at B	$n$ at C	$n$ at D	$n$ at E	$n$ at F	$n$ at G
odd	T	even	T	odd	even	odd

*Extension to all possible executions* This result was obtained by performing *one* execution completely, and then abstracting its outcome. Such a computation may of course not terminate, and it does not as wished describe all executions. The question is: how do we obtain even-odd information valid for *all possible computations*? A natural way is to simulate the computation, but to do the computation using the abstract values



$$\text{Abs} = \{\perp, \text{even}, \text{odd}, \top\}$$

instead of natural numbers, each representing a set of possible values of  $n$ ; and to ensure that all possible control flow paths are taken.

Doing this informally, we can see that if  $n$  is odd at program entry, it will always be even at points C and F, always odd at point E, sometimes even and sometimes odd at points B and D, and odd at G, provided control ever reaches G. Individual operations can be simulated by known properties of numbers, e.g.  $3n + 1$  is even if  $n$  is odd and odd if  $n$  is even, while  $n \div 2$  can be either even or odd.

Simulating the whole program is not as straightforward as simulating a single execution. The reason was mentioned before: execution over abstract values cannot in general be deterministic, since it must take account of all possible execution sequences on real data satisfying the abstract data description.

*Towards a less naive analysis procedure* The very earliest data-flow analysis algorithms amounted to glorified interpreters, and proceeded by executing the program symbolically, keeping a record of the desired flow information (abstract values) as the interpretation proceeded. Such algorithms, which in essence traced all possible control paths through the program, were very slow and often incorrect. They further suffered from a number of problems of semantic nature, for example difficulties in seeing how to handle nondeterminism due to tests with insufficient information to recognize their truth or falsity, convergence, and divergence of control paths, loops, and nontermination.

Better methods were soon developed to solve these problems, including

- putting a partial order on the abstract data values, so they can only change in the same direction during abstract interpretation, thus reducing termination problems
- storing flow information in a separate data structure, usually bound to program points (such as entry points to “basic blocks”, i.e. maximal linear program segments)
- constructing from the program a system of “data-flow equations”, one for each program point
- solving the data-flow equations (usually by computing their greatest fixpoint or least fixpoint).

Much more efficient algorithms were developed and some theoretical frameworks were developed to make the new methods more precise; [Hecht, 1977], [Kennedy, 1981], and [Aho *et al.*, 1986] contain good overviews.

None of the “classical” approaches to program analysis can, however, be said to be formally related to the semantics of the language whose programs were being analysed. Rather, they formalized and tightened up methods used in existing practice. In particular none of them was able to



include *precise execution as a special case of abstract interpretation* (albeit an uncomputable one). This was first done in [Cousot and Cousot, 1977b], the seminal paper relating abstract interpretation to program semantics.

## 2.2 Accumulating semantics for imperative programs

The approach of [Cousot and Cousot, 1977b] is appealing because of its generality: it expresses a large number of special program analyses in a common framework. In particular, this makes questions of safety (i.e. correctness) much easier to formulate and answer, and sets up a framework making it possible to relate and compare the precision of a range of different program analyses. It is solidly based in semantics, and precise execution of the program is included as a special case. This implies that program verification may also be based on the accumulating semantics, a theme developed further in [Cousot and Cousot, 1977a] and several subsequent works.

The ideas of [Cousot and Cousot, 1977b] have had a considerable impact on later work in abstract interpretation, for example [Mycroft, 1981], [Muchnick and Jones, 1981], [Burn *et al.*, 1986], [Donzeau-Gouge, 1981], [Nielson, 1982], [Nielson, 1984], [Mycroft, 1987].

### 2.2.1 Overview of the Cousot approach

The article [Cousot and Cousot, 1977b] begins by presenting an operational semantics for a simple flow chart language. It then develops the concept of what we call the *accumulating semantics* (the same as Cousots' static semantics and some others' collecting semantics). This associates with each program point the set of all memory stores that can ever occur when program control reaches that point, as the program is run on data from a given initial data space. It was shown in [Cousot and Cousot, 1977b] that a wide variety of flow analyses (but not all!) may be realized by finding finitely computable approximations to the accumulating semantics.

The (sticky) accumulating semantics maps program points to sets of program stores. The set  $\wp(\text{Store})$  of all sets of stores forms a *lattice* with set inclusion  $\subseteq$  as its partial order, so any two store sets  $A, B$  have least upper bound  $A \cup B$  and greatest lower bound  $A \cap B$ . The lattice  $\wp(\text{Store})$  is *complete*, meaning that any collection of sets of stores has a least upper bound in  $\wp(\text{Store})$ , namely its union.

Various approximations can be expressed by simpler lattices, connected to  $\wp(\text{Store})$  by an *abstraction* function  $\alpha : \wp(\text{Store}) \rightarrow \text{Abs}$  where  $\text{Abs}$  is a lattice of descriptions of sets of stores. Symbol  $\sqcup$  is usually used for the least upper bound operation on  $\text{Abs}$ ,  $\sqcap$  for the greatest lower bound, and  $\top, \perp$  for the least, resp. greatest elements of  $\text{Abs}$ .

An abstraction function is most often used together with a dual *concretization* function  $\gamma : \text{Abs} \rightarrow \wp(\text{Store})$ , and the two are required to satisfy natural conditions (given later).

For a one-variable program we could use as Abs the lattice with elements

$$\{\perp, \top, \text{even}, \text{odd}\},$$

where the abstraction of any nonempty set of even numbers is lattice element “even”, and the concretization of lattice element “even” is the set of all even numbers. Abstract interpretation may thus be thought of as executing the program over a lattice of imprecise but computable *abstract store descriptions* instead of the precise and uncomputable accumulating semantics lattice.

In practice computability is often achieved by using a *noetherian* lattice, i.e. one without infinite ascending chains. More general lattices can, however, be used, cf. the Cousots’ “widening” techniques, or the use of grammars to describe infinite sets finitely.

Let  $pp_0$  be the program’s initial program point and let  $pp$  be another program point. The set of store configurations that can be reached at program point  $pp$ , starting from a set  $S_0$  of possible initial stores, is defined by

$$\text{acc}_{pp} = \{s \mid (pp, s) = \text{next}^n((pp_0, s_0)) \text{ for some } s_0 \in S_0, n \geq 0\}$$

The accumulating semantics thus associates with each program point the set  $\text{acc}_p \subseteq \text{Store}$ .

### 2.2.2 Accumulating semantics of the example program

For the example program there is only one variable, so a set of stores has the form

$$\{[n \mapsto a_1], [n \mapsto a_2], [n \mapsto a_3], \dots\}$$

For notational simplicity we can identify this with the set  $\{a_1, a_2, a_3, \dots\}$  (an impossible simplification if the program has more than one variable). Given initial set  $S_0 = \{5\}$  the sets of stores reachable at each program point are:

$\text{acc}_A$	$\text{acc}_B$	$\text{acc}_C$	$\text{acc}_D$	$\text{acc}_E$	$\text{acc}_F$	$\text{acc}_G$
$\{5\}$	$\{5, 16, 8, 4, 2\}$	$\{16, 8, 4, 2\}$	$\{8, 4, 2, 1\}$	$\{5\}$	$\{16\}$	$\{1\}$

The following *data-flow* equations have a unique least fixpoint by completeness of  $\wp(\text{Store})$ , and it is easy to see that their fixpoint is exactly the tuple of sets of reachable stores as defined above.

$$\begin{aligned} \text{acc}_A &= S_0 \\ \text{acc}_B &= (\text{acc}_A \cup \text{acc}_D \cup \text{acc}_F) \cap \{n \mid n \in \{0, 1, 2, \dots\} \setminus \{1\}\} \\ \text{acc}_C &= \text{acc}_B \cap \{n \mid n \in \{0, 2, 4, \dots\}\} \end{aligned}$$

$$\begin{aligned}
 acc_D &= \{n \div 2 \mid n \in acc_C\} \\
 acc_E &= acc_B \cap \{n \mid n \in \{1, 3, 5, \dots\}\} \\
 acc_F &= \{3n + 1 \mid n \in acc_E\} \\
 acc_G &= (acc_A \cup acc_D \cup acc_F) \cap \{1\}
 \end{aligned}$$

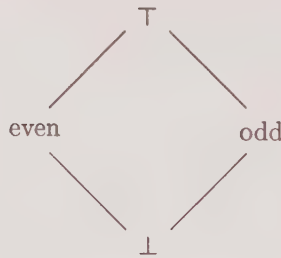
The equation set can be derived mechanically from the given program's syntax, e.g. as seen in [Cousot and Cousot, 1977b] or [Nielson, 1982].

### 2.2.3 Abstract interpretation of the example program

The *abstraction function*  $\alpha : \wp(\text{Store}) \rightarrow \text{Abs}$  below may be used to abstract a set of stores, where  $\text{Abs} = \{\perp, \text{even}, \text{odd}, \top\}$ :

$$\alpha(S) = \begin{cases} \perp & \text{if } S = \{\} \\ \text{even} & \text{if } S \subseteq \{0, 2, 4, \dots\} \\ \text{odd} & \text{if } S \subseteq \{1, 3, 5, \dots\} \\ \top & \text{else} \end{cases}$$

Defining  $\perp \sqsubseteq \text{even} \sqsubseteq \top$  and  $\perp \sqsubseteq \text{odd} \sqsubseteq \top$  makes Abs into a partially ordered set. Least upper and greatest lower bounds  $\sqcup, \sqcap$  exist so it is also a lattice.



Applying  $\alpha$  to the sets of reachable stores yields the following:

$abs_A$	$abs_B$	$abs_C$	$abs_D$	$abs_E$	$abs_F$	$abs_G$
odd	$\top$	even	$\top$	odd	even	odd

*Abstraction of the set of all runs* This method is still unsatisfactory for describing all computations since the value sets involved are unbounded and possibly infinite. But we may model the equations above by applying  $\alpha$  to the sets involved. The abstraction function  $\alpha$  just given is easily seen to be monotone, so set inclusion  $\subseteq$  in the world of actual computations is

modelled by  $\sqsubseteq$  in the world of simulated computations over Abs. Union is the least upper bound over sets, so it is natural to model  $\cup$  by  $\sqcup$ , and similarly to model  $\cap$  by  $\sqcap$ .

The arithmetic operations are faithfully modelled as follows, using familiar properties of natural numbers:

$$f_{n \div 2}(\text{abs}) = \begin{cases} \perp & \text{if abs} = \perp \\ \top & \text{else} \end{cases}$$

$$f_{3n+1}(\text{abs}) = \begin{cases} \perp & \text{if abs} = \perp, \text{ else} \\ \text{even} & \text{if abs} = \text{odd}, \text{ else} \\ \text{odd} & \text{if abs} = \text{even}, \text{ else} \\ \top & \text{if abs} = \top \end{cases}$$

This yields the following system of *approximate data-flow equations*, describing the program's behaviour on Abs:

$$\begin{aligned} \text{abs}_A &= \alpha(S_0) \\ \text{abs}_B &= (\text{abs}_A \sqcup \text{abs}_D \sqcup \text{abs}_F) \sqcap \top \quad (\text{"}\sqcap \top\text{" may be omitted}) \\ \text{abs}_C &= \text{abs}_B \sqcap \text{even} \\ \text{abs}_D &= f_{n \div 2}(\text{abs}_E) \\ \text{abs}_E &= \text{abs}_B \sqcap \text{odd} \\ \text{abs}_F &= f_{3n+1}(\text{abs}_E) \\ \text{abs}_G &= (\text{abs}_A \sqcup \text{abs}_D \sqcup \text{abs}_F) \sqcap \text{odd} \end{aligned}$$

*Remark* Here  $f_{n \div 2}$  and  $f_{3n+1}$  were defined ad hoc; a systematic way to define them will be seen in section 2.3.

The lattice Abs is also complete. The operators  $\sqcap$ ,  $\sqcup$ ,  $f_{n \div 2}$ , and  $f_{3n+1}$  are monotone, so the equation system has a (unique) least fixpoint. The abstraction function  $\alpha$  is easily seen to be monotone, so if it also were a homomorphism with respect to  $\cup$ ,  $\sqcup$  and  $\cap$ ,  $\sqcap$ , the least solution to the approximate flow equations would be exactly

$$\text{abs}_A = \alpha(\text{acc}_A), \dots, \text{abs}_G = \alpha(\text{acc}_G).$$

It is, however, *not* homomorphic since for example

$$\alpha(\{2\}) \sqcap \alpha(\{4\}) = \text{even} \neq \perp = \alpha(\{2\} \cap \{4\})$$

On the other hand the following *do* hold:

$$\begin{array}{llll}
\alpha(A) \sqcup \alpha(B) & = & \alpha(A \cup B) & f_{n \div 2}(\alpha(A)) \sqsupseteq \alpha(\{n \div 2 \mid n \in A\}) \\
\alpha(A) \sqcap \alpha(B) & \sqsupseteq & \alpha(A \cap B) & f_{3n+1}(\alpha(A)) \sqsupseteq \alpha(\{3n+1 \mid n \in A\})
\end{array}$$

Using these, it is easy to see by inspection of the two equation systems (more formally: a simple fixpoint induction) that their least fixpoints are related by

$$\begin{array}{ll}
\text{abs}_A & \sqsupseteq \alpha(\text{acc}_A), \\
\text{abs}_B & \sqsupseteq \alpha(\text{acc}_B), \\
& \vdots \\
\text{abs}_G & \sqsupseteq \alpha(\text{acc}_G)
\end{array}$$

The following is the iterative computation of the least fixpoint, assuming  $S_0 = \{5\}$ :

$\text{abs}_A$	$\text{abs}_B$	$\text{abs}_C$	$\text{abs}_D$	$\text{abs}_E$	$\text{abs}_F$	$\text{abs}_G$	iteration
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	0
odd	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	1
odd	odd	$\perp$	$\perp$	$\perp$	$\perp$	odd	2
odd	odd	$\perp$	$\perp$	odd	$\perp$	odd	3
odd	odd	$\perp$	$\perp$	odd	even	odd	4
odd	$\top$	$\perp$	$\perp$	odd	even	odd	5
odd	$\top$	even	$\perp$	odd	even	odd	6
odd	$\top$	even	$\top$	odd	even	odd	7, 8, ...

The conclusion is that  $n$  is always even at points C and F, and always odd at E and G.

### 2.2.4 An optimization using the results of the analysis

The flow analysis reveals that the program could be made somewhat more efficient by “unrolling” the loop after F. The reason is that tests “ $n \neq 1$ ” and “ $n$  even” must both be true in the iteration after F, so they need not be performed. The result is

```

while  $n \neq 1$  do if  $n$  even then  $n := n \div 2$  else  $n := (3 * n + 1) \div 2$ 
fi od

```

which avoids the two tests every time  $n$  is odd. In practice, one of the main reasons for doing abstract interpretation is to find out when such optimizing transformations may be performed.

### 2.2.5 Termination

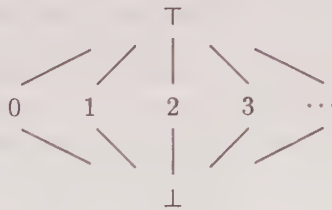
The least fixed point may be computed by beginning with  $[pp_1 \mapsto \perp, \dots, pp_m \mapsto \perp]$  (every program point is mapped to the least element of Abs), and



repeatedly replacing the value currently assigned to  $pp_i$  by the value of the right side of  $pp_i$ 's equation. By monotonicity of  $\sqcap$ ,  $f_{n+2}$  etc., these values can only grow or remain unchanged, so the iterations terminate provided the approximation lattice has no ascending chains of infinite height, as is the case here.

[Cousot and Cousot, 1977b] describe ways to achieve termination even when infinite chains exist, by inserting so-called *widening* operators in the data-flow equations at each junction point of a loop. To explain the basic idea consider the problem of finding the fixed point of a continuous function  $f$ . The usual Kleene iteration sequence is  $d_0 = \perp, \dots, d_{n+1} = f(d_n), \dots$  and is known to converge to the least fixed point of  $f$  but the sequence need not stabilize, i.e. it need not be the case that  $d_{n+1} = d_n$  for some  $n$ . To remedy this one may introduce a *widening* operator  $\nabla$  that dominates the least upper bound operation, i.e.  $d' \sqcup d'' \sqsubseteq d' \nabla d''$ , and such that the chain  $d_0 = \perp, \dots, d_{n+1} = d_n \nabla f(d_n)$  always stabilizes. This leads to overshooting the least fixed point but always gives a safe solution. By iterating down from the stabilization value (perhaps by using the technique of *narrowing*) one may then be able to recover some of the information lost.

*Constant propagation* This is an example of a lattice which is infinite but has finite height (three). It is used for detecting variables that do not vary, and has  $\text{Abs} = \{\top, \perp, 0, 1, 2, \dots\}$  where  $\perp \sqsubseteq n \sqsubseteq \top$  for  $n = 0, 1, 2, \dots$



The corresponding abstraction function is

$$\alpha(V) = \begin{cases} \perp & \text{if } V = \{ \} \\ n & \text{if } V = \{n\} \\ \top & \text{otherwise} \end{cases}$$

There also exist lattices in which all ascending chains have finite height, even though the lattice as a whole has unbounded vertical extent. An example: let  $\text{Abs} = (N, \geq)$ .

## 2.2.6 Safety: first discussion

The analysis of the Collatz-sequence program is clearly “safe” in the following sense: if control reaches point C then the value of  $n$  will be even, and similarly for the other program points and abstract values. Correctness (or

soundness) of the even-odd analysis for *all possible* programs and program points is also fairly easy to establish, given the close connection of the flow equations to those defining the accumulating semantics.

*Reachable program points* A similar but simpler *reachability* analysis (e.g. for dead code elimination) serves to illustrate a point concerning safety. It uses  $\text{Abs} = \{\top, \perp\}$  with  $\perp \sqsubseteq \top$  and abstraction function  $\alpha$  defined as follows (where  $a \in \text{Abs}$  and  $S \subseteq \text{Store}$ ):

$$\begin{array}{lll} \alpha(S) & = & \perp \quad \text{if } S = \{\} \quad \text{else } \top \\ f_{n+2}(a) & = & \perp \quad \text{if } a = \perp \quad \text{else } \top \\ f_{3n+1}(a) & = & \perp \quad \text{if } a = \perp \quad \text{else } \top \end{array}$$

Intuitively,  $\perp$  abstracts only the empty set of stores and so appropriately describes unreachable program points, while  $\top$  describes reachable program points. Computing the fixpoint as above we get

$$\frac{\text{abs}_A \quad \text{abs}_B \quad \text{abs}_C \quad \text{abs}_D \quad \text{abs}_E \quad \text{abs}_F \quad \text{abs}_G}{\top \quad \top \quad \top \quad \top \quad \top \quad \top \quad \top}$$

This might be thought to imply that *all* program points including G are reachable, regardless of the initial value of  $n$ . On the other hand, reachability of G for input  $n$  implies termination, and it is a well-known open question whether the program does in fact terminate for all  $n$ .

A more careful analysis reveals that  $\perp$  at program point  $pp$  represents “ $pp$  cannot be reached”, while  $\top$  represents “ $pp$  might be reached” and so does not necessarily imply termination. The example shows that we must examine the questions of correctness and safety more carefully, which we now proceed to do.

## 2.3 Correctness and safety

In this and the remaining parts of section 2, we describe informally several different approaches to formulating safety and correctness, and discuss some advantages and disadvantages. A more detailed domain-based framework will be set up in section 3.

### 2.3.1 Desirable properties of the abstract value set Abs

In order to model the accumulating semantics equations, Abs could be a *complete lattice*: a set with a partial order  $\sqsubseteq$ , with least upper and greatest lower bounds  $\sqcup$  and  $\sqcap$  to model  $\cup$  and  $\cap$ , and such that any collection of sets of stores has a least upper bound in Abs. Note: any lattice of finite height is complete. In the following we sometimes write  $a \sqsupseteq a'$  in place of  $a' \sqsubseteq a$ .

### 2.3.2 Desirable properties of the abstraction function

Intuitively “even” represents the set of all even numbers. This viewpoint is made explicit in [Cousot and Cousot, 1977b] by relating complete lattices  $\text{Conc}$  and  $\text{Abs}$  to each other by a pair  $\alpha, \gamma$  of *abstraction and concretization* functions with types

$$\begin{aligned}\alpha &: \text{Conc} \rightarrow \text{Abs} \\ \gamma &: \text{Abs} \rightarrow \text{Conc}\end{aligned}$$

In the even-odd example above the lattice of concrete values is  $\text{Conc} = \wp(\text{Store})$ , and the natural concretization function is

$$\begin{aligned}\gamma(\perp) &= \{\} \\ \gamma(\text{even}) &= \{0, 2, 4, \dots\} \\ \gamma(\text{odd}) &= \{1, 3, 5, \dots\} \\ \gamma(\top) &= \{0, 1, 2, 3, \dots\} = \mathbb{N}\end{aligned}$$

Cousot and Cousot impose natural conditions on  $\alpha$  and  $\gamma$  (satisfied by the examples):

1.  $\alpha$  and  $\gamma$  are monotonic
2.  $\forall a \in \text{Abs}, a = \alpha(\gamma(a))$
3.  $\forall c \in \text{Conc}, c \sqsubseteq_{\text{Conc}} \gamma(\alpha(c))$

For the accumulating semantics, larger abstract values represent larger sets of stores by condition 1. Condition 2 is natural, and condition 3 says that  $S \subseteq \gamma(\alpha(S))$  for any  $S \subseteq \text{Store}$ .

The conditions can be summed up as:  $(\alpha, \gamma)$  form a Galois insertion of  $\text{Abs}$  into  $\wp(\text{Store})$ , a special case of an adjunction in the sense of category theory. It is easy to verify the following:

**Lemma 2.3.1.** *If conditions 1–3 hold, then*

- $\forall c \in \text{Conc}, a \in \text{Abs}: c \sqsubseteq_{\text{Conc}} \gamma(a)$  if and only if  $\alpha(c) \sqsubseteq_{\text{Abs}} a$ , and
- $\alpha$  is continuous.

Thus the abstract flow equations converge to a fixpoint. If  $\alpha$  is semihomomorphic on union, intersection, and base functions, then the abstract flow equations’ fixpoint will be pointwise larger than or equal to the abstraction of the fixpoint of the accumulating semantics’ equations.

Again, note that stores are unordered, so  $\alpha$  and  $\gamma$  need only preserve the subset ordering. The more complex situation that arises when modelling nonflat domains is investigated in [Mycroft and Nielson, 1983].

### 2.3.3 Safety: second discussion

Recalling the program of section 2.2, we can define the solution  $(\text{abs}_A, \dots, \text{abs}_G) \in \text{Abs}^7$  to the abstract flow equations to be *safe* with

respect to the accumulating semantics  $(acc_A, \dots, acc_G) \in \wp(\text{Store})^7$  if the reachable sets of stores are represented by the abstract values:

$$\begin{aligned} acc_A &\subseteq \gamma(abs_A), \\ acc_B &\subseteq \gamma(abs_B), \\ &\vdots \\ acc_G &\subseteq \gamma(abs_G) \end{aligned}$$

This is easy to verify for the even-odd abstraction given before.

Returning to the question raised after the “reachable program points” example, we see that safety at point  $G$  only requires that  $acc_G \subseteq \gamma(abs_G)$ , i.e. that every store that can reach  $G$  appears in  $\gamma(abs_G)$ . This also holds if  $acc_G$  is empty, so  $\gamma(abs_G) = \top$  does *not* imply that  $G$  is reachable in any actual computation. For any program point  $X$ ,  $abs_X = \perp$  implies  $acc_X = \{\}$ , which signifies that control cannot reach  $X$ . Thus abstract value  $\perp$  can be used to eliminate dead code.

*Safe approximation of base functions* Consider base function  $op : \mathbb{IN} \rightarrow \mathbb{IN}$ , and extend it, by “pointwise lifting” to sets of numbers, yielding  $\wp op : \wp(\mathbb{IN}) \rightarrow \wp(\mathbb{IN})$  where

$$\wp op(N) = \{op(n) \mid n \in N\}$$

Suppose  $\alpha, \gamma$  satisfy conditions 1–3. It is natural to define  $\mathbf{op} : \text{Abs} \rightarrow \text{Abs}$  to be a *safe approximation* to  $op$  if the following holds for all  $N \subseteq \mathbb{IN}$ :

$$\wp op(N) \subseteq \gamma(\mathbf{op}(\alpha(N)))$$

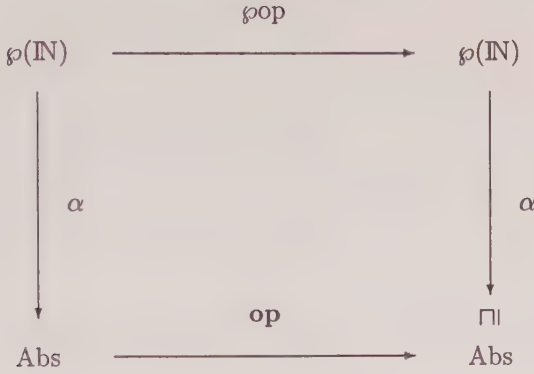
or, diagrammatically:

$$\begin{array}{ccc} \wp(\mathbb{IN}) & \xrightarrow{\wp op} & \wp(\mathbb{IN}) \\ \downarrow \alpha & & \uparrow \gamma \\ \text{Abs} & \xrightarrow{\mathbf{op}} & \text{Abs} \end{array}$$

By the conditions and lemma this is equivalent to

$$\alpha(\wp op(N)) \subseteq \mathbf{op}(\alpha(N))$$

corresponding to the diagram



Intuitively, for any subset  $N \subseteq \mathbb{N}$ , applying the induced abstract operation **op** to the abstraction of  $N$  represents at least all the values obtainable by applying **op** to members of  $N$ .

*Induced approximations to base functions* We now show how the best possible approximation **op** can be extracted from **op** (at least in principle, although perhaps not computably: a more detailed discussion appears in section 3.4). Recall that smaller elements of **Abs** abstract smaller sets of concrete values and so are less approximate, i.e. more precise, descriptions.

**Lemma 2.3.2.** *Given  $\alpha : \wp(\mathbb{N}) \rightarrow \text{Abs}$  and  $\gamma : \text{Abs} \rightarrow \wp(\mathbb{N})$  satisfying the three conditions above, the operator induced by **op** is, by definition, **op** :  $\text{Abs} \rightarrow \text{Abs}$  where*

$$\mathbf{op} = \alpha \circ \wp \text{op} \circ \gamma$$

*Then **op** is the most precise function on **Abs** satisfying  $\alpha(\wp \text{op}(N)) \sqsubseteq \mathbf{op}(\alpha(N))$  for all  $N$ .*

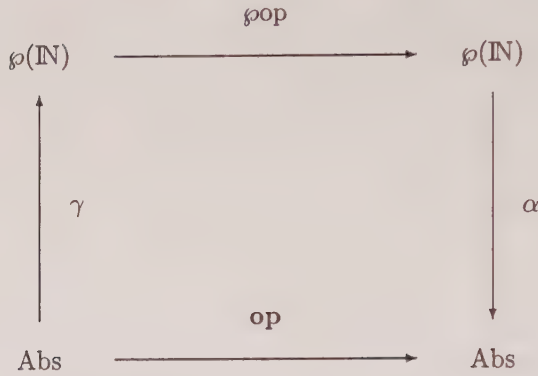
**Proof.** Suppose  $f : \text{Abs} \rightarrow \text{Abs}$  with  $\alpha(\wp \text{op}(N)) \sqsubseteq f(\alpha(N))$  for all  $N$ . Then for any  $a$ ,

$$\mathbf{op}(a) = \alpha(\wp \text{op}(\gamma(a))) \sqsubseteq f(\alpha(\gamma(a))) = f(a)$$

■

The definition of **op** as a diagram is as follows





For example, if  $\text{op}(n) = n \div 2$  then  $\text{op}$  is  $f_{n \div 2}$  as seen above, e.g.

$$\begin{aligned}
 \text{op}(\perp) &= \alpha(\{n \div 2 \mid n \in \gamma(\perp)\}) = \alpha(\{\}) = \perp \\
 \text{op}(\text{even}) &= \alpha(\{n \div 2 \mid n \in \gamma(\text{even})\}) = \alpha(\{0, 1, 2, \dots\}) = \top
 \end{aligned}$$

Unfortunately the definition of  $\text{op}$  does not necessarily give a terminating algorithm for computing it, even if  $\text{op}$  is computable. In practice the problem is solved by approximating from above, i.e. choosing  $\text{op}$  to give values in  $\text{Abs}$  that may be larger (less informative) than implied by the above equation. We will go deeper into this in section 3.5.

*A local condition for safe approximation of transitions* A safety condition on one-step transitions can be formulated analogously. Define for any two control points  $p, q$  the function  $\text{next}_{p,q}: \wp(\text{Store}) \rightarrow \wp(\text{Store})$ :

$$\text{next}_{p,q}(S) = \{s' \mid (q, s') = \text{next}((p, s)) \text{ for some } s \in S\}$$

This is the earlier transition function, extended to include all transitions from  $p$  to  $q$  on a set of stores. Exactly as above we can define the abstract transition function *induced* by  $\alpha$  and  $\gamma$  to be

$$\text{next}_{p,q} = \alpha \circ \text{next}_{p,q} \circ \gamma$$

This is again the most precise function satisfying

$$\text{next}_{p,q}(\alpha(S)) \subseteq \alpha(\text{next}_{p,q}(S))$$

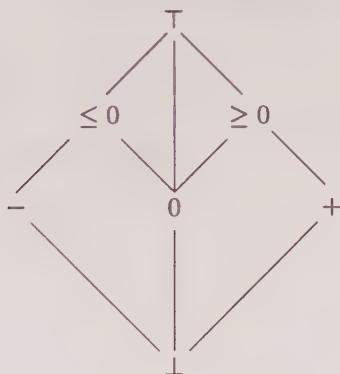
for all  $S$ .

### 2.3.4 An example: the rule of signs

Consider the abstract values  $+$ ,  $-$  and  $0$  with the natural concretization function

$$\begin{aligned}
 \gamma(0) &= \{0\} \\
 \gamma(+) &= \{1, 2, 3, \dots\} \\
 \gamma(-) &= \{-1, -2, -3, \dots\}
 \end{aligned}$$

This can be made into a complete lattice by adding greatest lower and least upper bounds in various ways. Assuming  $\sqcap, \sqcup$  should model  $\cap, \cup$  respectively, the following is obtained



with

$$\begin{aligned}
 \gamma(\perp) &= \{\} \\
 \gamma(\geq 0) &= \{0, 1, 2, 3, \dots\} \\
 \gamma(\leq 0) &= \{0, -1, -2, -3, \dots\} \\
 \gamma(\top) &= \{\dots, -2, -1, 0, 1, 2, 3, \dots\} = \mathbb{Z}
 \end{aligned}$$

and abstraction function

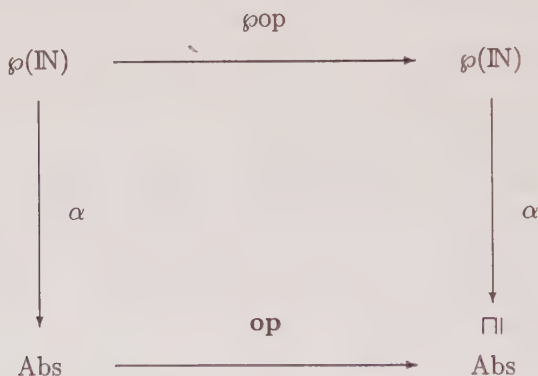
$$\alpha(S) = \begin{cases} \perp & \text{if } S = \{\} \text{ else} \\ + & \text{if } S \subseteq \{1, 2, 3, \dots\} \text{ else} \\ \geq 0 & \text{if } S \subseteq \{0, 1, 2, 3, \dots\} \text{ else} \\ - & \text{if } S \subseteq \{-1, -2, -3, \dots\} \text{ else} \\ \leq 0 & \text{if } S \subseteq \{0, -1, -2, -3, \dots\} \text{ else} \\ \top & \end{cases}$$

The induced approximation for operator  $+: \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$  is

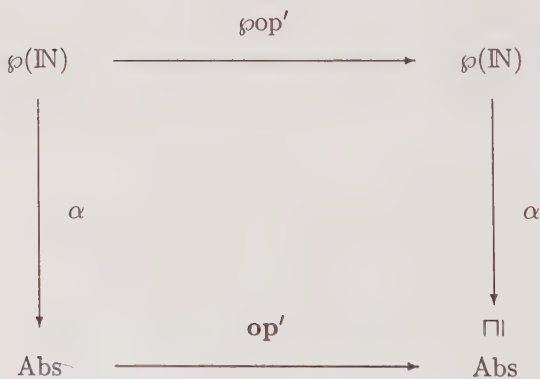
$+$ '	$\perp$	$-$	$0$	$+$	$\geq 0$	$\leq 0$	$\top$
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
$-$	$\perp$	$-$	$-$	$\top$	$\top$	$-$	$\top$
$0$	$\perp$	$-$	$0$	$+$	$\geq 0$	$\leq 0$	$\top$
$+$	$\perp$	$\top$	$+$	$+$	$+$	$\top$	$\top$
$\geq 0$	$\perp$	$\top$	$\geq 0$	$+$	$\geq 0$	$\top$	$\top$
$\leq 0$	$\perp$	$-$	$\leq 0$	$\top$	$\top$	$\leq 0$	$\top$
$\perp$	$\perp$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$

### 2.3.5 Composition of safety diagrams

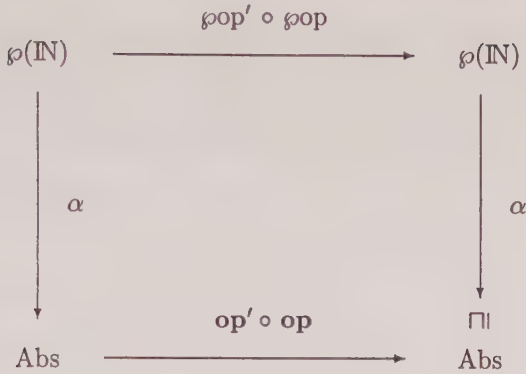
Suppose we have two diagrams for the safe approximation of two base functions  $\text{op}$  and  $\text{op}'$ :



and



It is easy to see that  $\text{op}' \circ \text{op}$  is a safe approximation to  $\wp \text{op}' \circ \wp \text{op}$ , so the two may be composed:



On the other hand the diagrams for the *induced* approximations to base functions *cannot* be so composed, since the best approximation to  $\wp\text{op}' \circ \wp\text{op}$  may be better than the composition of the best approximations to  $\wp\text{op}$  and  $\wp\text{op}'$ . (This is precisely because  $\alpha$  is a semihomomorphism, not a homomorphism.) For a concrete example, let  $\text{op}$  and  $\text{op}'$  respectively describe the effects of the two assignments

$$n := 4 * n + 2; \quad n := n \div 2$$

Then

$$\alpha(\wp\text{op}' \circ \wp\text{op}(\{0, 1, 2, \dots\})) = \alpha(\{1, 3, 5, \dots\}) = \text{odd}$$

whereas

$$\text{op}' \circ \text{op} (\alpha(\{0, 1, 2, \dots\})) = \text{op}'(\text{even}) = \top.$$

## 2.4 Scott domains, lattice duality, and meet versus join

*Relation to Scott-style domains* The partial order  $\sqsubseteq$  on  $\text{Abs}$  models the set inclusion order  $\subseteq$  used for  $\wp(\text{Store})$  in the accumulating semantics. In abstract interpretation, larger elements of  $\text{Abs}$  correspond to *more approximate* descriptions, so if  $a \sqsubseteq a'$  then  $a'$  describes a *larger set* of concrete values. For example, “even” describes any set of even numbers, and  $\top$  describes the set of all numbers.

In contrast, Scott domains as used in denotational semantics use an ordering by “information content”, where a larger domain element describes *a single value that is more completely calculated*. During a computation  $\perp$  means “not yet calculated”, intuitively a slot to be filled in later with the a more complete value. The appearance of  $\perp$  in a program’s final result signifies “was never filled in”, and so represents nontermination (at least in languages with eager evaluation).

A value in a Scott domain represents perhaps incomplete knowledge about a *single* program value, for example a finite part of an infinite function  $f$ . The partial order  $f \sqsubseteq f'$  signifies that  $f'$  is more completely defined than  $f$ , and that  $f'$  agrees with  $f$  wherever it is defined.  $\top$ , if used at all, indicates inconsistent values.

Clearly this order is not the same as the one used in abstract interpretation, and the difference is more than just one of duality.

*Least or greatest fixpoints?* The literature on data-flow analysis as used in compilers [Aho *et al.*, 1986], [Hecht, 1977], [Kennedy, 1981] often uses abstract value lattices which are dual to the ones we consider, so larger elements represent more precise descriptions rather than more approximate. This is mainly a matter of taste, but has the consequence that *greatest* fixpoints are computed instead of least ones, and that the  $\cup$  and  $\cap$  of the accumulating semantics are modelled by  $\sqcap$  and  $\sqcup$ , respectively. We prefer least fixpoints due to their similarity to those naturally used in defining the accumulating semantics.

*Should  $\sqcup$  or  $\sqcap$  be used on converging paths?* We have argued that  $\sqcup$  naturally models the effect of path convergence because it corresponds to  $\cup$  in the accumulating semantics. On the other hand, there exist abstract interpretations that are *not* approximations to the accumulating semantics, and for some of these path convergence is properly modelled by  $\sqcap$ . To see this, consider the two dependence analyses mentioned in section 1.1. For analysis I, path convergence should be modelled by  $\sqcup$  since a variable dependence is to be recorded if it occurs along *at least one* path. For analysis II it should be modelled by  $\sqcap$  since a dependence is recorded only if it occurs along *all* paths. So the choice between  $\sqcup$  and  $\sqcap$  on converging paths is just another incarnation of the modality distinction encountered in section 1.

## 2.5 Abstract values viewed as relations or predicates

The accumulating semantics binds to each program point a set of stores. Suppose the program's variables are  $V_1, \dots, V_n$ , so a store is an element of  $\text{Store} = \{V_1, \dots, V_n\} \rightarrow \text{Value}$ . In the examples above there was only one variable, so a set of stores was essentially a set of values, which simplified the discussion considerably. The question arises: how can we abstract a set of stores when  $n > 1$ ?

### 2.5.1 Independent attribute analyses

Suppose value sets are abstracted by  $\alpha_{val} : \wp(\text{Value}) \rightarrow A$ . The *independent attribute method* models a set of stores  $S$  at program point  $p$  by mapping each variable  $V_i$  to an abstraction of the set of values it takes in all the stores of  $S$ . This abstract value is thus independent of all other variables, hence the term "independent attribute". Thus  $\{[X \mapsto 1, Y \mapsto 2], [X \mapsto$



3,  $Y \mapsto 1\}$ ) would be modelled by  $[X \mapsto \text{odd}, Y \mapsto \top]$ .

Formally, we model

$$S \in \wp(\text{Store}) = \wp(\{V_1, \dots, V_n\} \rightarrow \text{Value})$$

by a function

$$\text{abs}_p \in \text{Abs} = \{V_1, \dots, V_n\} \rightarrow A$$

The store abstraction function  $\alpha_{sto} : \wp(\text{Store}) \rightarrow \text{Abs}$  is defined by

$$\alpha_{sto}(S) = [V_i \mapsto \alpha_{val}(\{s(V_i) \mid s \in S\})]_{i=1, \dots, n}$$

For example, consider an even-odd analysis of a program with variables  $X, Y, Z$ . The independent attribute method would abstract a set of two stores as follows:

$$\begin{aligned} \alpha_{sto}(\{[X \mapsto 1, Y \mapsto 2, Z \mapsto 1], [X \mapsto 2, Y \mapsto 2, Z \mapsto 1]\}) &= \\ [X \mapsto \alpha_{val}(\{1, 2\}), Y \mapsto \alpha_{val}(\{2\}), Z \mapsto \alpha_{val}(\{1\})] &= \\ [X \mapsto \top, Y \mapsto \text{even}, Z \mapsto \text{odd}] \end{aligned}$$

The independent attribute method abstracts each variable independently of all others, and so allows “cross over” effects. An example:

$$\begin{aligned} \alpha(\{[X \mapsto 1, Y \mapsto 1], [X \mapsto 2, Y \mapsto 2]\}) &= [X \mapsto \top, Y \mapsto \top] = \\ \alpha(\{[X \mapsto 1, Y \mapsto 2], [X \mapsto 2, Y \mapsto 1]\}) \end{aligned}$$

This loses information about relationships between  $X$ ’s and  $Y$ ’s values, e.g. whether or not they always have the same parity.

### 2.5.2 Relational analyses

*Relations and predicates* Abstract value  $\text{abs}_p$  is an abstraction of the set of stores  $\text{acc}_p$ , so the question arises as to how to represent it by a lattice element. An approach used in [Cousot and Cousot, 1977b], [Cousot and Cousot, 1977a] is to describe  $\text{acc}_p$  and its approximations  $\text{abs}_p$  by predicate calculus formulas. For instance, the two-store set  $\{[X \mapsto 1, Y \mapsto 1], [X \mapsto 2, Y \mapsto 2]\}$  above could be approximately described by the formula,

$$(\text{odd}(X) \wedge \text{odd}(Y)) \vee (\text{even}(X) \wedge \text{even}(Y))$$

More generally, suppose  $\text{Store} = \{V_1, \dots, V_n\} \rightarrow \text{Value}$ . Clearly  $\text{Store}$  is isomorphic to  $\text{Value}^n$ , the set of all  $n$ -tuples of values. Thus any set of stores, i.e. any element of  $\wp(\text{Store})$  can be interpreted as a set of  $n$ -tuples.

For example, store set  $\{[X \mapsto 1, Y \mapsto 1], [X \mapsto 2, Y \mapsto 2]\}$  corresponds to  $\{(1,1), (2,2)\}$ . Thus a store set is essentially a set of  $n$ -tuples or, in other words, an  $n$ -ary *predicate* or *relation* on values.

For program point  $p$ , the accumulating semantics defines relation  $\text{acc}_p(v_1, \dots, v_n)$  to be true just in the case that  $(v_1, \dots, v_n)$  is a tuple of values which can occur at  $p$  in one or more computations on the given initial input data. This is the weakest possible relation among variables that always holds at point  $p$ .

*Relational analyses* These use more sophisticated methods to approximate  $\wp(\text{Store})$ , which can give more precise information. Examples of practically motivated program analysis problems that require relational information include aliasing analysis in Pascal, the recognition of possible substructure sharing in Lisp or Prolog, and interference analysis.

For an example not naturally represented by independent attributes, suppose we wish to find out which of a program's variables always assume the same value at a given program point  $p$ . A suitable abstraction of a set of stores is a *partition*  $\pi_p$  that divides the program's variables into equivalence classes, so any one class of  $\pi_p$  contains all variables that have the same value at  $p$ . The effect of an assignment such as " $p : X := Y; \text{goto } q$ " is that  $\pi_q$  is obtained from  $\pi_p$  by removing  $X$  from its previous equivalence class and adding it to  $Y$ 's class.

*Intensional versus extensional descriptions* Above we represented store set

$$\{[X \mapsto 1, Y \mapsto 1], [X \mapsto 2, Y \mapsto 2]\}$$

by the binary relation  $\{(1,1), (2,2)\}$ , and approximated it by the superset  $\{(x,y) \mid x \text{ and } y \text{ are both even or both odd}\}$ , denoted by the predicate calculus formula

$$(\text{odd}(X) \wedge \text{odd}(Y)) \vee (\text{even}(X) \wedge \text{even}(Y))$$

The view of "predicate as a set of tuples" and "predicate as a formula" is exactly the classical distinction between the *extensional* and the *intensional* views of a predicate.

Descriptions by predicate calculus formulas must of necessity be only approximate, since there are only countably many formulas but uncountably many sets of stores (if we assume an infinite variable value set). In terms of predicate calculus formulas, for each program point  $p$  the appropriate formulation of a safe approximation is that  $\text{acc}_p$  *logically implies*  $\text{abs}_p$ . In terms of sets of  $n$ -tuples: each  $\text{acc}_p$  is a subset of the set of all tuples satisfying  $\text{abs}_p$ .

### 2.5.3 Abstract interpretation and predicate transformers

The new view of the accumulating semantics is: given a program and a predicate describing its input data, the accumulating semantics maps every program point to the smallest *relation among variables that holds whenever control reaches that point*.

From this viewpoint, the function  $\text{next}_{p,q} : \wp(\text{Store}) \rightarrow \wp(\text{Store})$  is clearly the *forward predicate transformer* [Dijkstra, 1976] associated with transitions from  $p$  to  $q$ . Further,  $\text{acc}_p$  is clearly *the strongest postcondition* holding at program point  $p$  over all computations on input data satisfying the program's input precondition.

Program verification amounts to proving that each  $\text{acc}_p$  logically implies a user-supplied program assertion for point  $p$ . Note, however, that this abstract interpretation framework says nothing at all about program termination. This approach is developed further in [Cousot and Cousot, 1977a] and their subsequent works.

*Backwards analyses* All this can easily be dualised: the *backward predicate transformer*  $\text{next}_{p,q}^{-1} : \wp(\text{Store}) \rightarrow \wp(\text{Store})$  is just the inverse of  $\text{next}_{p,q}$ , and given a *program postcondition* one may find the *weakest precondition* on program input sufficient to imply the postcondition at termination. For the simple imperative language, a backward accumulating semantics is straightforward to construct. For the example program

```

A: while  $n \neq 1$  do
  B: if  $n$  even
    then (C:  $n := n \div 2$ ; D: )
    else (E:  $n := 3 * n + 1$ ; F: )
  fi
od
G:

```

the appropriate equations are

$$\begin{aligned}
 \text{acc}_A &= (\{1\} \cap \text{acc}_G) \cup (\{0, 2, 3, 4, \dots\} \cap \text{acc}_B) \\
 \text{acc}_B &= (\text{acc}_C \cap \text{Evens}) \cup (\text{acc}_E \cap \text{Odds}) \\
 \text{acc}_C &= \{n \mid n \div 2 \in \text{acc}_D\} \\
 \text{acc}_D &= (\{1\} \cap \text{acc}_G) \cup (\{0, 2, 3, 4, \dots\} \cap \text{acc}_B) \\
 \text{acc}_E &= \{n \mid 3n + 1 \in \text{acc}_F\} \\
 \text{acc}_F &= (\{1\} \cap \text{acc}_G) \cup (\{0, 2, 3, 4, \dots\} \cap \text{acc}_B) \\
 \text{acc}_G &= S_{\text{final}}
 \end{aligned}$$

where  $\text{acc}_p$  is the set of all stores at point  $p$  that cause control to reach point G with a final store in  $S_{\text{final}}$ .

Such a backward accumulating semantics can, for example, provide a basis for an analysis that detects the set of states that may lead to an error. More generally backwards analyses (although not the one shown here) may provide a basis for “future sensitive” analysis such as *live variables*, where variable  $X$  is “semantically live” at point  $p$  if there is a computation sequence starting at  $p$  and later referencing  $X$ ’s value. This is approximated by:  $X$  is “syntactically live” if there is a program path from  $p$  to a use of  $X$ ’s value. Section 3 contains an example of live variable analysis for functional programs.

Many analysis problems can be solved by either a forwards or a backwards analysis. There can, however, be significant differences in efficiency.

*Backwards analysis of functional programs* The backward accumulating semantics is straightforward for imperative programs, partly because of its close connections with the well-studied weakest preconditions [Dijkstra, 1976], and because the state transition function is monadic. It is semantically less well understood, however, for functional programs, where recent works include [Hughes, 1987], [Dybjer, 1987], [Wadler and Hughes, 1987], and [Nielson, 1989]. Natural connections between backwards analyses and continuation semantics are seen in [Nielson, 1982] and [Hughes, 1987].

## 2.6 Important points from earlier sections

In the above we have employed a rather trivial programming language so as to motivate and illustrate one way to approximate real computations by computations over a domain of abstract values: Cousot’s accumulating semantics. Before proceeding to abstract interpretation of more interesting languages we recapitulate what has been learned so far.

- Computations in the abstract world are at best *semihomomorphic* models of corresponding computations in the world of actual values.
- *Safety* of an abstract interpretation is analogous to *reliability* of a numerical analyst’s results: the obtained results must always lie within specified confidence intervals (usually “one-sided intervals” in the case of program analysis).
- To obtain safe results for specific applications it is essential to understand the interpretation of the abstract values and their relation to actual computational values. One example is *modality*, e.g. “all computations” versus “some computations”.
- Abstract values often do not contain enough information to determine the outcome of tests, so abstract interpretation must achieve the effect of simulating a *set* of real computations.
- Computations on *complete lattices* of abstract values appropriately model computations on real values.
- The partial order on these lattices expresses the degree of precision in an approximate description, and is *quite different* from the traditional



Scott-style ordering based on filling in incomplete information.

- Termination can be achieved by choosing lattices without infinite ascending chains.
- *Best* approximations to real computations exist in principle, but may be uncomputable.
- There are close connections between the “accumulating semantics” and the predicates and predicate transformers (both forwards and backwards) used in program verification.

## 2.7 Towards generalizing the Cousot framework

Abstract interpretation is a semantics-based approach to program analysis, but so far we have only dealt with a single, rather trivial language. Rather than redo the same work for every new language, we set the foundations for developing a general framework based on *denotational semantics*. This is a widely used and rather general formalism for defining the semantics of programming languages (see [Schmidt, 1986; Stoy, 1977; Gordon, 1979; Nielson and Nielson, 1992a]). Other possibilities include axiomatic and structural operational semantics [Plotkin, 1981], and natural semantics [Kahn, 1987], [Nielson and Nielson, 1992a]. They are also general frameworks, but ones in which few applications to abstract interpretation have been developed (although operational semantics seems especially promising).

The approach will be developed stepwise. First, a denotational semantics is given for essentially the simple imperative language seen before. This is then factored into two stages, into a *core semantics* and an *interpretation*. The interpretation specifies the details relevant to a specific (standard or abstract) interpretation of the program’s values and operations, and the core semantics specifies those parts of the semantics that are to be used in the same way for all interpretations. It is then shown how, given a fixed core semantics, interpretations may be partially ordered with respect to “degree of abstractness”, and it is shown that a concrete interpretation’s execution results are always compatible with those of more abstract interpretations. This provides a basis for formally proving the safety of an analysis, for example by showing that a given abstract interpretation is an abstraction of the accumulating interpretation. The last step is to describe briefly a way to generalize this approach to denotational definitions of other languages; this gives a bridge to the development of section 3.

Earlier papers using this approach include [Donzeau-Gouge, 1981], [Nielson, 1982], [Jones and Mycroft, 1986].

*Denotational semantics* has three basic principles:

1. Every syntactic phrase in a program has a *meaning*, or *denotation*.
2. Denotations are well-defined mathematical objects (often higher-order functions).



3. The meaning of a compound syntactic phrase is a mathematical combination of the meanings of its immediate subphrases.

The last assumption is often called *compositionality* or, according to Stoy, *the denotational assumption*. Phrase meanings are most often given by expressions in the typed lambda calculus, although other possibilities exist. A denotational language definition consists of the following parts:

- a collection of *domain definitions*, to be used as types for the lambda expressions used to define phrase meanings
- a collection of *semantic functions*, usually one for each syntactic phrase type
- a collection of *semantic rules* defining them, expressing the meanings of syntactic phrases in terms of the meanings of their substructures, usually by lambda expressions.

*A tiny denotational language definition* For an example, consider a language of **while**-programs with abstract syntax

$$\begin{aligned} c : \text{Cmd} &::= x := e \mid c ; c' \\ &\quad \mid \text{if } e \text{ then } c \text{ else } c' \mid \text{while } e \text{ do } c \\ e : \text{Exp} &::= x \mid \text{Constant} \mid \text{op}(e_1, \dots, e_n) \end{aligned}$$

where  $x$  is assumed to range over a set  $\text{Var}$  of variables,  $\text{op}$  is a basic operation (e.g.  $+$ ,  $-$ ,  $*$ ,  $\div$ ), and  $\text{Constant}$  denotes any member of a not further specified set  $\text{Value}$  of values. The part of a denotational semantics relevant to commands could be as follows, where the traditional “semantic parentheses”  $\llbracket$  and  $\rrbracket$  enclose syntactic arguments. In the following  $\text{Store}$  and  $\text{Value}$  are as before except that for concreteness we specify  $\text{Value}$  as the set of numbers. Each can be thought of as a “flat” Scott domain with  $d \sqsubseteq d'$  iff  $d = \perp$  or  $d = d'$ .

Specifying the semantics of a **while** loop requires (as usual) evaluating the least fixpoint over domain  $\text{Store} \rightarrow \text{Store}$ , ordered “pointwise”:  $s \sqsubseteq s'$  iff  $s(x) \sqsubseteq s'(x)$  for all variables  $x$ .

#### Domain definitions

$$\begin{aligned} s : \text{Store} &= \text{Var} \rightarrow \text{Value} \\ \text{Value} &= \text{Number} \end{aligned}$$

#### Types of semantic and auxiliary functions

$$\begin{aligned} C : \text{Cmd} &\rightarrow \text{Store} \rightarrow \text{Store} \\ E : \text{Exp} &\rightarrow \text{Store} \rightarrow \text{Value} \end{aligned}$$

#### Semantic rules

$$\begin{aligned} C[x := e] &= \lambda s . s[x \mapsto E[e] s] \\ C[c ; c'] &= \lambda s . C[c'] \\ C[\text{if } e \text{ then } c \text{ else } c'] &= \lambda s . E[e] s \neq 0 \rightarrow C[c] s, C[c'] s \end{aligned}$$

$$C[\text{while } e \text{ do } c] = \text{fix } \lambda\phi . \lambda s . E[e] \sigma \neq 0 \rightarrow \phi(C[c] s), s$$

Note that all the rules are compositional.

### 2.7.1 Factoring a denotational semantics

The principle of compositionality provides an ideal basis for generalizing the denotational semantics framework to allow alternate, nonstandard interpretations in addition to the “standard” semantics defining the meanings of programs. The idea is to decompose a denotational language definition into two parts:

- a *core semantics*, containing semantic rules and their types, but using some uninterpreted domain names and function symbols, and
- an *interpretation*, filling out the missing definitions of domain names and function symbols.

The interpretation is clearly a many-sorted algebra. Examples include the “standard interpretation” defining normal program execution, an “accumulating interpretation” analogous to the accumulating semantics of section 2, and as well more approximate and effectively computable interpretations suitable for program analysis, e.g. for parity analysis.

*Scott domains versus complete lattices* In denotational semantics the denoted values are nearly always elements of “domains” in the sense of Dana Scott and others. These are cpo’s (complete partial orders with  $\perp$ ), usually required to be algebraic. For material on domains see the list of references in the beginning of this chapter.

On the other hand for abstract interpretation purposes, it is usual to use complete lattices for reasons mentioned earlier. There is no basic conflict here since cpo’s include complete lattices as a special case. On the other hand, the interpretation of the partial order is somewhat different in semantics than in abstract interpretation (as mentioned in section 2.4), so some care must be taken. This matter is further addressed in section 3.

*An example factorized semantics* For the imperative language above we obtain the following, where domains *Sto* and *Val*, and functions *assign*, *seq*, *cond*, *while*, are unspecified:

#### Domain definitions

$$\begin{aligned} M_{Cmd} &= \text{Sto} \rightarrow \text{Sto} \\ M_{Exp} &= \text{Sto} \rightarrow \text{Val} \\ \text{Sto, Val:} &\text{ unspecified} \end{aligned}$$

#### Types of semantic and auxiliary functions

$$\begin{aligned} C &: \text{Cmd} \rightarrow M_{Cmd} \\ E &: \text{Exp} \rightarrow M_{Exp} \\ \text{assign} &: \text{Var} \times M_{Exp} \rightarrow M_{Cmd} \end{aligned}$$

$$\begin{aligned}
\text{seq} & : M_{Cmd} \times M_{Cmd} \rightarrow M_{Cmd} \\
\text{cond} & : M_{Exp} \times M_{Cmd} \times M_{Cmd} \rightarrow M_{Cmd} \\
\text{while} & : M_{Exp} \times M_{Cmd} \rightarrow M_{Cmd}
\end{aligned}$$

### Semantic rules

$$\begin{aligned}
C[x := e] & = \text{assign}(x, E[e]) \\
C[c ; c'] & = \text{seq}(C[c], C[c']) \\
C[\text{if } e \text{ then } c \text{ else } c'] & = \text{cond}(E[e], C[c], C[c']) \\
C[\text{while } e \text{ do } c] & = \text{while}(E[e], C[c])
\end{aligned}$$

*The standard interpretation* This is  $I_{std} = (\text{Val}, \text{Sto}; \text{assign}, \text{seq}, \text{cond}, \text{while})$ , defined by

### Domains

$$\begin{aligned}
\text{Val} & = \text{Number (the flat cpo)} \\
\text{Sto} & = \text{Var} \rightarrow \text{Val}
\end{aligned}$$

### Function definitions

$$\begin{aligned}
\text{assign} & = \lambda(x, m_e) . \lambda s . s[x \mapsto m_e s] \\
\text{seq} & = \lambda(m_{1c}, m_{2c}) . m_{2c} \circ m_{1c} \\
\text{cond} & = \lambda(m_e, m_{1c}, m_{2c}) . \lambda s . m_e s \neq 0 \rightarrow m_{1c} s, m_{2c} s \\
\text{while} & = \lambda(m_e, m_c) . \text{fix } \lambda\phi . \lambda s . m_e s \neq 0 \rightarrow \phi(m_c s), s
\end{aligned}$$

## 2.7.2 The even-odd interpretation

With the current machinery a general formulation of the even-odd analysis of section 2.2.3 may be given as our first nonstandard interpretation:

$$I_{parity} = (\text{Val}, \text{Sto}; \text{assign}, \text{seq}, \text{cond}, \text{while})$$

where Val, etc. are given by:

### Domains

$$\begin{aligned}
\text{Val} & = \{\perp, \text{even}, \text{odd}, \top\} \text{ with partial order} \\
& \quad \perp \sqsubseteq \text{even} \sqsubseteq \top \text{ and } \perp \sqsubseteq \text{odd} \sqsubseteq \top \\
\text{Sto} & = \text{Var} \rightarrow \text{Val}
\end{aligned}$$

### Function definitions

$$\begin{aligned}
\text{assign} & = \lambda(x, m_e) \lambda s . s[x \mapsto m_e s] \\
\text{seq} & = \lambda(m_{1c}, m_{2c}) . m_{2c} \circ m_{1c} \\
\text{cond} & = \lambda(m_e, m_{1c}, m_{2c}) \lambda s . m_{1c} s \sqcup m_{2c} s \\
\text{while} & = \lambda(m_e, m_c) . \text{fix } \lambda\phi \lambda s . \phi(m_c s) \sqcup s
\end{aligned}$$

### Remarks

1. This is an *independent attribute* approximation: a set of stores is modelled by mapping its variables' values independently to elements of Val.

2. For the conditional, no attempt is made to simulate the test. Instead, the best description fitting both the then and else branches is produced.
3. The fixpoint clearly models the one in the standard interpretation, again without simulating the test. Termination is assured since  $\text{Sto}$  has finite height and any one program has a finite number of variables.

### 2.7.3 The accumulating semantics as an interpretation

The accumulating semantics seen earlier was only given by example. With the current machinery a general formulation may be given:  $\mathbf{I}_{acc} = (\text{Val}, \text{Sto}; \text{assign}, \text{seq}, \text{cond}, \text{while})$  where  $\text{Val}$ , etc. are given by

#### Domains

$\text{Val}$	$=$	$\wp(\text{Number})$	
$\text{Sto}$	$=$	$\text{Storeset}$	typical element $S$
$\text{Storeset}$	$=$	$\wp(\text{Var} \rightarrow \text{Val})$	a set of stores

#### Function definitions

$\text{assign}$	$=$	$\lambda(x, m_e) . \lambda S . \{s[x \mapsto v] \mid s \in S \text{ and } v \in m_e\{s\}\}$
$\text{seq}$	$=$	$\lambda(m_{1c}, m_{2c}) . m_{2c} \circ m_{1c}$
$\text{cond}$	$=$	$\lambda(m_e, m_{1c}, m_{2c}) . \lambda S .$ $m_{1c} (\{s \in S \mid 0 \notin m_e\{s\}\}) \cup m_{2c} (\{s \in S \mid 0 \in m_e\{s\}\})$
$\text{while}$	$=$	$\lambda(m_e, m_c) . \text{fix } \lambda\phi . \lambda S .$ $\{s \in S \mid m_e s = 0\} \cup \phi(m_c \{s \in S \mid m_e s \neq 0\})$

Here the denotation of  $\mathbf{C}[\text{Cmd}]$  has been “lifted” from  $\text{Sto} \rightarrow \text{Sto}$  to  $\wp(\text{Sto}) \rightarrow \wp(\text{Sto})$ , so it now transforms a set of current states into the set of possible next states. To relate this to the earlier accumulating semantics of section 2.2, consider for example the program fragment “ $\text{C} : n := n \div 2; \text{D}$ ” of section 2.2.2. Then

$$\mathbf{C}[n := n \div 2]_{acc_C} = acc_D$$

In general,  $\mathbf{C}[c]$  realizes the same transformation on store sets as defined by the data-flow equations for command  $c$ .

## 2.8 Proving safety by logical relations

### 2.8.1 Safety from a denotational viewpoint

Suppose we are given two interpretations

$$\mathbf{I} = (\text{Val}, \text{Sto}; \text{assign}, \text{seq}, \text{cond}, \text{while})$$

and

$$\mathbf{I}' = (\text{Val}', \text{Sto}'; \text{assign}', \text{seq}', \text{cond}', \text{while}')$$

and a pair of abstraction functions

$$\beta = (\beta_{val} : \text{Val} \rightarrow \text{Val}', \beta_{sto} : \text{Sto} \rightarrow \text{Sto}')$$

where  $\beta_{val}$  and  $\beta_{sto}$  are monotone. We write this as  $\beta: \mathbf{I} \rightarrow \mathbf{I}'$ . By definition,  $s \sqsubseteq s'$  holds iff  $s(X) \sqsubseteq s'(X)$  for all variables  $X \in \text{Var}$ .

**Definition 2.8.1.**  $\beta: \mathbf{I} \rightarrow \mathbf{I}'$  is *safe* if for all  $c \in \text{Cmd}$  and  $s \in \text{Sto}$

$$\beta_{sto}(C_I[[c]]\ s) \sqsubseteq C_{I'}[[c]]\ (\beta_{sto}\ s)$$

where  $C_I : \text{Sto} \rightarrow \text{Sto}$  is the semantic function obtained using `assign`, `seq`, etc. and  $C_{I'}$  is analogous but using `assign'`, `seq'`, etc.

Recall that  $a \sqsubseteq a'$  signifies that  $a'$  is a more approximate description than  $a$ . This definition says that the result of computing in the “real world” and then abstracting the resulting store gives a result that is safely approximable by first abstracting the real world’s initial store, and then computing entirely in the “abstract world”. It is thus simply a reformulation in denotational terms of the earlier condition on safe approximation of transitions:

$$\alpha(\text{next}_{p,q}(S)) \sqsubseteq \text{next}_{p,q}(\alpha(S))$$

where  $p$  and  $q$  are (resp.) the entry and exit points of command  $c$  (and  $\alpha$  plays the role of  $\beta$ ).

## 2.8.2 A sufficient local condition for safety

While pleasingly general, this definition is unfortunately global: it quantifies over all commands  $c$ . It would be strongly desirable to have *local* conditions on `assign`, `seq`, etc. sufficient to guarantee safety in the sense above. This can be done, but requires first setting up a bit of descriptive machinery. The problem is that denotational definitions use higher order functions and cartesian products, whereas the earlier conditions for safety were developed only for first-order domains.

The solution we present is an instance of *logical relations*, an approach to relating values in different but similarly structured domains that will be developed further in section 3.3.

Suppose we have two interpretations  $\mathbf{I}$  and  $\mathbf{I}'$  of our simple imperative core semantics, related by  $\beta = (\beta_{val} : \text{Val} \rightarrow \text{Val}', \beta_{sto} : \text{Sto} \rightarrow \text{Sto}')$  where  $\beta_{val}$  and  $\beta_{sto}$  are again monotone. Our goal is to see how to extend  $\beta$  to apply to all domains built up from those of  $\mathbf{I}$  and  $\mathbf{I}'$ . Suppose further that domain  $A$  is built by  $\times$  and  $\rightarrow$  from the domains of  $\mathbf{I}$ , and a corresponding domain  $A'$  is built in the same way from the domains of  $\mathbf{I}'$ . We define the binary relation  $a \leq_\beta a'$ , which will hold whenever  $a' \in A'$  is a safe approximation of the corresponding element  $a \in A$ .

**Definition 2.8.2.** Suppose  $\beta = \beta_{val} : \text{Val} \rightarrow \text{Val}'$  and  $v \in \text{Val}, v' \in \text{Val}'$ . Then

1.  $v \leq_\beta v'$  if and only if  $\beta_{val}(v) \sqsubseteq v'$  and similarly for  $\beta = \beta_{sto} : \text{Sto} \rightarrow \text{Sto}'$ .
2. Let  $(a, b) \in A \times B$ ,  $(a', b') \in A' \times B'$  and  $\beta_A : A \rightarrow A'$ ,  $\beta_B : B \rightarrow B'$  be monotone. Then

$$(a, b) \leq_\beta (a', b')$$

if and only if

$$a \leq_{\beta_A} a' \text{ and } b \leq_{\beta_B} b'.$$



3. Let  $f : A \rightarrow B$ ,  $g' : A' \rightarrow B'$  and  $\beta_A : A \rightarrow A'$ ,  $\beta_B : B \rightarrow B'$  be monotone. Then

$f \leq_\beta g$  if and only if

$$\forall a \in A \forall a' \in A' (a \leq_{\beta_A} a' \text{ implies } fa \leq_{\beta_B} ga').$$

This notation allows an alternate characterization of safety.

**Lemma 2.8.3.**  $\beta : \mathbf{I} \rightarrow \mathbf{I}'$  is safe if and only if  $C_I[c] \leq_\beta C_{I'}[c]$  for all  $c \in \text{Cmd}$ .

**Proof.** “If”: by 3,  $C_I[c] \leq_\beta C_{I'}[c]$  holds if and only if  $C_I[c]s \leq_{\beta_{sto}} C_{I'}[c]s'$  whenever  $s \leq_{\beta_{sto}} s'$ . In particular we have  $s \leq_\beta \beta_{sto}(s)$  hence

$$\beta_{sto}(C_I[c]s) \subseteq C_{I'}[c]\beta_{sto}(s)$$

so  $\beta$  is safe (as defined before).

“Only if”: if  $\beta$  is safe and  $s \leq_\beta s'$  then

$$\beta_{sto}(C_I[c]s) \subseteq C_{I'}[c](\beta_{sto}s) \subseteq C_{I'}[c]s'$$

by monotonicity of  $C_{I'}[c]$  (easily verified). ■

It is now natural to extend the definition of  $\leq_\beta$  to allow comparison of interpretations. Given this, we are finally ready to define a local safety condition which implies global safety. The proof is by a straightforward induction on program syntax.

**Definition 2.8.4.** Let  $\beta : \mathbf{I} \rightarrow \mathbf{I}'$  be defined as above. Then  $\mathbf{I} \leq_\beta \mathbf{I}'$  if and only if

$$\begin{array}{lll} \text{assign} & \leq_\beta & \text{assign}' \\ \text{seq} & \leq_\beta & \text{seq}' \\ \text{cond} & \leq_\beta & \text{cond}' \\ \text{while} & \leq_\beta & \text{while}'. \end{array}$$

**Theorem 2.8.5.**  $\beta : \mathbf{I} \rightarrow \mathbf{I}'$  is safe if  $\mathbf{I} \leq_\beta \mathbf{I}'$ .

A straightforward generalization of these ideas to arbitrary denotational definitions provides a very general framework for program analysis by interpreting programs over nonstandard domains, and gives a way to show that one abstract interpretation is a refinement of and compatible with another. This observation is the starting point for the development in the section below.

For a very simple example, the following is easy to show. Its significance is that the accumulating semantics is a faithful extension of the standard semantics.

**Lemma 2.8.6.** Let  $\mathbf{I}_{std} = (Val, Sto; assign, seq, cond, while)$  and  $\mathbf{I}_{acc} = (Val', Sto'; assign', seq', cond', while')$  be the standard and accumulating semantics. Then  $\mathbf{I}_{std} \leq_{\beta} \mathbf{I}_{acc}$ , where  $\beta_{val}(v) = \{v\}$  for  $v \in Val$  and  $\beta_{sto}(s) = \{s\}$  for  $s \in Sto$ .

### 3 Abstract interpretation using a two-level meta-language

In the previous section we have given a survey of many concepts in abstract interpretation. We have stressed that abstract interpretation should be *generally applicable* to programs in a wide class of languages, that it should always produce *correct* properties, and that it should always *terminate*.

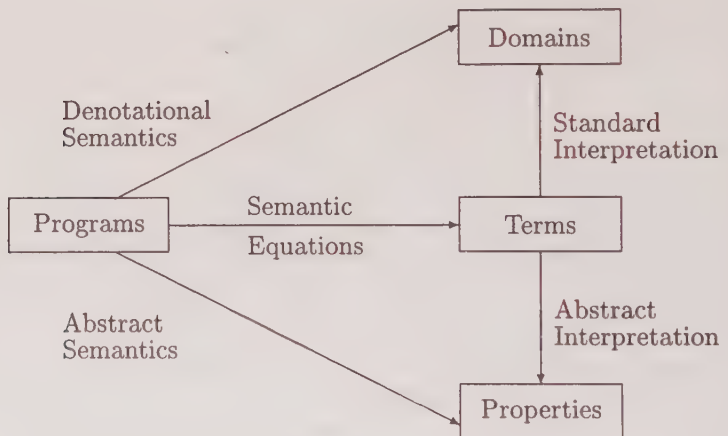


Fig. 1. The role of the metalanguage

To ensure the *general applicability* of abstract interpretation we adopt the framework of denotational semantics. Most modern approaches to denotational semantics stress the role of a formal metalanguage in which the semantics is defined. So rather than regarding denotational semantics as directly mapping programs to mathematical domains one regards denotational semantics as factored through the metalanguage. This is illustrated by the upper half of Figure 1. First one uses the semantic equations to expand programs into terms in the metalanguage and then one interprets the terms in the metalanguage as elements in the mathematical domains used in denotational semantics.

Following our informal presentation of a parameterized semantics in section 2.7 we take a similar factored approach to abstract interpretation.

This is illustrated in the lower half of Figure 1 which furthermore stresses that the semantic equations are the same. The semantic equations thus correspond to the core semantics of section 2.7. From the point of view of developing a general theory, the focus will be on the metalanguage but we trust that the reader will be able to see that the development is of wider applicability than just the simple metalanguage considered here. Its syntax is defined in section 3.1 and we give example interpretations (i.e. semantics) in section 3.2.

*Correctness* of abstract interpretation will be our guide throughout the development. In section 3.3 we therefore give a structural definition of correctness relations between interpretations thereby extending the development surveyed in section 2.8. As an example we define correctness relations between the standard interpretation and one of the abstract interpretations defined in section 3.2.

Closely related to the question of correctness is the question of whether best *induced* property transformers exist over the abstract domains. We treat this in section 3.4 and we consider the easier case of relating abstract interpretations to one another as well as the harder case of relating an abstract interpretation to the standard interpretation. Induced property transformers need not terminate but are none the less useful as guides in determining the degree of approximation that will be needed to ensure termination.

The *termination* aspect motivates the study in section 3.5 of coarser versions of the induced property transformers which have the advantage of leading to analyses that will always terminate. section 3.6 concludes by mentioning some generalizations that are possible [Nielson, 1989] and by discussing some issues that have not yet been incorporated into this treatment.

### 3.1 Syntax of metalanguage

Most metalanguages for denotational semantics are based on some version of the  $\lambda$ -calculus. Depending on the kind of mathematical foundations used for denotational semantics the metalanguage may be without explicit types or it may have explicit types. We shall not pay great attention to this difference and in many instances the various algorithms for polymorphic type inference may be used to introduce types into an untyped notation. As our starting point we thus assume that our metalanguage is a small typed  $\lambda$ -calculus.

**Definition 3.1.1.** The typed  $\lambda$ -calculus has types  $t \in T$  and expressions  $e \in E$  given by

$$\begin{aligned} t &::= A_i \mid t \times t \mid t \rightarrow t \\ e &::= f_i[t] \mid \langle e, e \rangle \mid \text{fst } e \mid \text{snd } e \mid \\ &\quad \lambda x_i[t].e \mid e(e) \mid x_i \mid \text{fix } e \mid \text{if } e \text{ then } e \text{ else } e \end{aligned}$$

Concerning types we have base types  $A_i$  where  $i$  ranges over a countable index set (say  $I$ ) and we have product and function space. Here  $\times$  binds more tightly than  $\rightarrow$  and both associate to the right. We shall not specify the details of the countable index set but we shall assume that we have booleans  $A_{\text{bool}}$  (also written  $\text{Bool}$ ), integers  $A_{\text{int}}$  (also written  $\text{Int}$ ) and other useful base types. However, nothing precludes us from having a base type  $A_{\text{sto}}$  of machine stores and if the store contains just two values, say an integer and a boolean, we may write  $A_{\text{int} \times \text{bool}}$  for  $A_{\text{sto}}$ . (The difference between types like  $A_{\text{int} \times \text{bool}}$  and  $A_{\text{int}} \times A_{\text{bool}}$  will become clear in section 3.2.)

Concerning expressions we have basic expressions  $f_i[t]$  of type  $t$  where again  $i$  ranges over a countable index set. (This index set need not be the same as the one used for the  $A_i$  above but whenever we need to name it we shall use the same symbol  $I$  as above.) Again we expect to have familiar basic expressions like the truth values  $f_{\text{true}}[\text{Bool}]$  and  $f_{\text{false}}[\text{Bool}]$  (also written  $\text{true}[\text{Bool}]$  and  $\text{false}[\text{Bool}]$  or just  $\text{true}$  and  $\text{false}$ ), integers like  $f_0[\text{Int}]$  (also written  $0[\text{Int}]$  or just  $0$ ), and simple operations like equality  $f_{=}[\text{Int} \times \text{Int} \rightarrow \text{Bool}]$  (also written  $=[\text{Int} \times \text{Int} \rightarrow \text{Bool}]$  or just  $=$ ). Much as for the  $A_i$  nothing prevents us from writing, for example,  $f_{\lambda x. \lambda y. x=y+y}[\text{Int} \rightarrow \text{Int} \rightarrow \text{Bool}]$  in order to clarify the intended meaning of some basic expression. The remaining constructs for expressions are pairing, selection of components,  $\lambda$ -abstraction, application, variables, fixed points, and conditional. We shall assume that application binds more tightly than  $\text{fst}$ ,  $\text{snd}$ , and  $\text{fix}$ .

### 3.1.1 The use of underlining

We shall postpone the discussion about well-formedness of expressions in the typed  $\lambda$ -calculus because the syntax is not yet in a form that will suit our purpose: to prescribe a systematic approach to separating a denotational semantics into its core part and its interpretation part (to use the terminology of section 2.7). To motivate this we recall from section 2 that for a given programming language or example semantics there are some constructs that we might wish to interpret in different ways in different analyses whereas there are other constructs that we might as well interpret in the same way in all analyses. To indicate this distinction in a precise way we shall use the convention that *underlined* constructs are those that should have the freedom to be interpreted freely.

Beginning with the types we might consider a syntax as given by

$$t ::= A_i \mid \underline{A_i} \mid t \times t \mid t \rightarrow t$$

so that we would use  $\underline{A_{\text{int}}}$  (also written  $\underline{\text{Int}}$ ) instead of  $A_{\text{int}}$  whenever the integers are used in a context where we would like to perform abstract interpretation upon their values. Thus if we want to consider the store of an imperative programming language as a base type we will always use  $\underline{A_{\text{sto}}}$  rather than  $A_{\text{sto}}$ . If we want to consider a structured version of the store



where we have a fixed set  $A_{\text{ide}}$  of identifiers and a fixed set  $A_{\text{val}}$  of values we shall use  $A_{\text{ide}} \rightarrow A_{\text{val}}$  rather than, for example,  $A_{\text{ide}} \rightarrow A_{\text{val}}$  or  $A_{\text{ide}} \rightarrow A_{\text{val}}$ .

However, this notation for types does not allow us to illustrate all the points we will need for a general theory of abstract interpretation although it would suffice for formalizing the development in section 2.7. Examples include the discussion of forward versus backward analyses and the discussion of independent attribute versus relational methods. To cater for this we propose the syntax

$$t ::= A_i \mid \underline{A_i} \mid t \times t \mid \underline{t \times t} \mid t \rightarrow t \mid \underline{t \rightarrow t}$$

and we shall use the phrase *two-level types* for these. This will turn out to be a bit too liberal for our abilities so we shall need to impose various well-formedness conditions upon the types, but in order to motivate them it is best to postpone this until they are needed for the technical development. In actual applications there might well be the need for distinguishing between various occurrences of  $\times$  and  $\rightarrow$  and one might then allow a notation like  $\times_i$  and  $\rightarrow_i$  where  $i$  ranges over some index set. However, as the theory hardly changes we shall leave this extension to the reader.

Turning to the expressions, a simple solution would be to keep the syntax of expressions as given in Definition 3.1.1 with the understanding that the types  $t$  in the basic expressions  $f_i[t]$  now range over the larger set of two-level types. However, this is not quite in the spirit of the typed  $\lambda$ -calculus as we now have types without corresponding constructors and destructors. We shall therefore adopt a more comprehensive syntax of expressions by extending the use of underlining to the expressions.

**Definition 3.1.2.** The two-level  $\lambda$ -calculus has types  $t \in T$  and expressions  $e \in E$  given by

$$\begin{aligned} t &::= A_i \mid \underline{A_i} \mid t \times t \mid \underline{t \times t} \mid t \rightarrow t \mid \underline{t \rightarrow t} \\ e &::= f_i[t] \mid \langle e, e \rangle \mid \underline{\langle e, e \rangle} \mid \text{fst } e \mid \underline{\text{fst } e} \mid \text{snd } e \mid \underline{\text{snd } e} \mid \\ &\quad \lambda x_i[t].e \mid \underline{\lambda x_i[t].e} \mid e(e) \mid \underline{e(e)} \mid x_i \mid \\ &\quad \text{fix } e \mid \underline{\text{fix } e} \mid \text{if } e \text{ then } e \text{ else } e \mid \underline{\text{if } e \text{ then } e \text{ else } e} \end{aligned}$$

Here the intention is that if, for example,  $e_1$  is of type  $t_1$  and  $e_2$  is of type  $t_2$  then  $\langle e_1, e_2 \rangle$  will be of type  $t_1 \times t_2$  and  $\underline{\langle e_1, e_2 \rangle}$  will be of type  $\underline{t_1 \times t_2}$  and similarly for the other operators. We do not have two versions of  $f_i[t]$  as  $f_i[t]$  simply is a basic expression of the type indicated, nor do we have two versions of  $x_i$  as  $x_i$  simply is a placeholder for a “pointer” to the enclosing  $\lambda x_i$  or  $\underline{\lambda x_i}$ . In this notation an operation  $\text{seq}$  for sequencing two commands operating on a store  $\text{Sto}$  might be defined by

$$\text{seq} = \lambda x_1[\text{Sto} \rightarrow \text{Sto}]. \lambda x_2[\text{Sto} \rightarrow \text{Sto}]. \underline{\lambda x_{\text{sto}}[\text{Sto}]. x_2(x_1(x_{\text{sto}}))}$$

It will have type

$$(\text{Sto} \rightarrow \text{Sto}) \rightarrow (\text{Sto} \rightarrow \text{Sto}) \rightarrow (\text{Sto} \rightarrow \text{Sto})$$



and may be used as in  $C[\mathbf{c}_1; \mathbf{c}_2] = \text{seq}(C[\mathbf{c}_1])(C[\mathbf{c}_2])$ .

### 3.1.2 Combinators

The motivation behind the use of underlining was to separate the more “dynamic” constructs that need to be interpreted freely from the more “static” constructs whose interpretation never changes. Unfortunately the two-level  $\lambda$ -calculus is not in a form that makes this sufficiently easy. The problem is the occurrence of free variables and especially those bound by  $\lambda$ . This is not a novel problem and solutions have been found:

- When interpreting the typed  $\lambda$ -calculus in arbitrary cartesian closed categories one studies certain combinators (“categorical combinators”) whose interpretation in a cartesian closed category is rather straightforward [Curien, 1986].
- When implementing functional languages one often transforms programs to combinator form before performing graph reduction.

This motivates:

**Definition 3.1.3.** The two-level metalanguage has types  $t \in T$  and expressions  $e \in E$  given by

$$\begin{aligned} t &::= A_i \mid \underline{A_i} \mid t \times t \mid \underline{t \times t} \mid t \rightarrow t \mid \underline{t \rightarrow t} \\ e &::= f_i[t] \mid \langle e, e \rangle \mid \text{Tuple}(e, e) \mid \text{fst } e \mid \text{Fst } e \mid \text{snd } e \mid \text{Snd } e \\ &\quad \mid \lambda x_i[t]. e \mid \text{Curry } e \mid e(e) \mid \text{Apply}(e, e) \mid x_i \mid \text{Id}[t] \mid e \square e \\ &\quad \mid \text{Const}[t] e \mid \text{fix } e \mid \text{Fix } e \mid \text{if } e \text{ then } e \text{ else } e \mid \text{If}(e, e, e) \end{aligned}$$

Here we have retained those expression constructs that were not underlined, we have replaced the underlined expression constructs by combinators, and we have added the new combinators  $\text{Id}[t]$ ,  $\square$ , and  $\text{Const}[t]$ . We shall regard application as binding more tightly than the prefixed operators ( $\text{fst}$ ,  $\text{Fst}$ ,  $\text{snd}$ ,  $\text{Snd}$ ,  $\text{Curry}$ ,  $\text{Const}[t]$ ,  $\text{fix}$ , and  $\text{Fix}$ ). The intention with the combinators may be clarified by

$$\begin{aligned} \text{Tuple}(e_1, e_2) &\equiv \lambda x_1. \langle \underline{e_1(x_1)}, \underline{e_2(x_1)} \rangle \\ \text{Fst } e &\equiv \lambda x_1. \underline{\text{fst } e(x_1)} \\ \text{Snd } e &\equiv \lambda x_1. \underline{\text{snd } e(x_1)} \\ \text{Curry } e &\equiv \lambda x_1. \lambda x_2. \underline{e(\langle x_1, x_2 \rangle)} \\ \text{Apply}(e_1, e_2) &\equiv \lambda x_1. \underline{e_1(x_1)}(\underline{e_2(x_1)}) \\ \text{Id}[t] &\equiv \lambda x_1. x_1 \\ e_1 \square e_2 &\equiv \lambda x_1. \underline{e_1(e_2(x_1))} \\ \text{Const}[t](e) &\equiv \lambda x_1. e \\ \text{Fix } e &\equiv \lambda x_1. \underline{\text{fix } e(x_1)} \\ \text{If}(e_1, e_2, e_3) &\equiv \lambda x_1. \underline{\text{if } e_1(x_1) \text{ then } e_2(x_1) \text{ else } e_3(x_1)} \end{aligned}$$

This should be rather familiar to anyone who knows a bit of categorical logic or a bit of a functional language like FP. In this notation the sequencing

$tenv \vdash_{c1,c2} f_i[t] : t$	if $\vdash_{c1} t$
$tenv \vdash_{c1,c2} e_1 : t_1$	$tenv \vdash_{c1,c2} e_2 : t_2$
$tenv \vdash_{c1,c2} \langle e_1, e_2 \rangle : t_1 \times t_2$	
$tenv \vdash_{c1,c2} e_1 : t \rightarrow t_1$	$tenv \vdash_{c1,c2} e_2 : t \rightarrow t_2$
$tenv \vdash_{c1,c2} \text{Tuple}(e_1, e_2) : t \rightarrow t_1 \times t_2$	
if $\vdash_{c1} (t \rightarrow t_1) \rightarrow (t \rightarrow t_2) \rightarrow (t \rightarrow t_1 \times t_2)$	
$tenv \vdash_{c1,c2} e : t_1 \times t_2$	
$tenv \vdash_{c1,c2} \text{fst } e : t_1$	
$tenv \vdash_{c1,c2} e : t \rightarrow t_1 \times t_2$	
$tenv \vdash_{c1,c2} \text{Fst } e : t \rightarrow t_1$	
if $\vdash_{c1} (t \rightarrow t_1 \times t_2) \rightarrow (t \rightarrow t_1)$	
$tenv \vdash_{c1,c2} e : t_1 \times t_2$	
$tenv \vdash_{c1,c2} \text{snd } e : t_2$	
$tenv \vdash_{c1,c2} e : t \rightarrow t_1 \times t_2$	
$tenv \vdash_{c1,c2} \text{Snd } e : t \rightarrow t_2$	
if $\vdash_{c1} (t \rightarrow t_1 \times t_2) \rightarrow (t \rightarrow t_2)$	
$tenv[t/x_i] \vdash_{c1,c2} e : t'$	if $\vdash_{c2} t$
$tenv \vdash_{c1,c2} \lambda x_i[t].e : t \rightarrow t'$	
$tenv \vdash_{c1,c2} e : t \times t' \rightarrow t''$	
$tenv \vdash_{c1,c2} \text{Curry } e : t \rightarrow t' \rightarrow t''$	
if $\vdash_{c1} (t \times t' \rightarrow t'') \rightarrow (t \rightarrow t' \rightarrow t'')$	
$tenv \vdash_{c1,c2} e_1 : t' \rightarrow t$	$tenv \vdash_{c1,c2} e_2 : t'$
$tenv \vdash_{c1,c2} e_1(e_2) : t$	
$tenv \vdash_{c1,c2} e_1 : t \rightarrow t' \rightarrow t''$	$tenv \vdash_{c1,c2} e_2 : t \rightarrow t'$
$tenv \vdash_{c1,c2} \text{Apply}(e_1, e_2) : t \rightarrow t''$	
if $\vdash_{c1} (t \rightarrow t' \rightarrow t'') \rightarrow (t \rightarrow t') \rightarrow t \rightarrow t''$	
$tenv \vdash_{c1,c2} x_i : t$	if $\vdash_{c2} t \wedge tenv(x_i) = t$

Table 1. Well-formedness of expressions (part 1)

operator `seq` used above simply is

$$\text{seq} = \lambda x_1[\text{Sto} \rightarrow \text{Sto}]. \lambda x_2[\text{Sto} \rightarrow \text{Sto}]. x_2 \square x_1$$

and thus there hardly is any need to name it.

To complete the definition of the two-level metalanguage we must explain when expressions are well-formed. We have already said that we shall need to impose conditions on the types as we go along and the well-formedness condition will be influenced by this although in a rather in-

$tenv \vdash_{c1,c2} Id[t] : t \rightarrow t$	$if \vdash_{c1} t \rightarrow t$
$\frac{tenv \vdash_{c1,c2} e_1 : t_0 \rightarrow t_1 \quad tenv \vdash_{c1,c2} e_2 : t_1 \rightarrow t_2}{tenv \vdash_{c1,c2} e_2 \square e_1 : t_0 \rightarrow t_2}$	$if \vdash_{c1} (t_1 \rightarrow t_2) \rightarrow (t_0 \rightarrow t_1) \rightarrow (t_0 \rightarrow t_2)$
$\frac{tenv \vdash_{c1,c2} e : t'}{tenv \vdash_{c1,c2} Const[t] e : t \rightarrow t'}$	$if \vdash_{c1} t' \rightarrow t \rightarrow t'$
$\frac{tenv \vdash_{c1,c2} e : t \rightarrow t}{tenv \vdash_{c1,c2} fix e : t}$	
$\frac{tenv \vdash_{c1,c2} e : t \rightarrow t' \rightarrow t'}{tenv \vdash_{c1,c2} Fix e : t \rightarrow t'}$	$if \vdash_{c1} (t \rightarrow t' \rightarrow t') \rightarrow (t \rightarrow t')$
$\frac{tenv \vdash_{c1,c2} e_1 : Bool \quad tenv \vdash_{c1,c2} e_2 : t \quad tenv \vdash_{c1,c2} e_3 : t}{tenv \vdash_{c1,c2} if e_1 then e_2 else e_3 : t}$	
$\frac{tenv \vdash_{c1,c2} e_1 : t \rightarrow Bool \quad tenv \vdash_{c1,c2} e_2 : t \rightarrow t' \quad tenv \vdash_{c1,c2} e_3 : t \rightarrow t'}{tenv \vdash_{c1,c2} If\langle e_1, e_2, e_3 \rangle : t \rightarrow t'}$	$if \vdash_{c1} (t \rightarrow Bool) \rightarrow (t \rightarrow t') \rightarrow (t \rightarrow t') \rightarrow (t \rightarrow t')$

Table 2. Well-formedness of expressions (part 2)

direct way. As we shall see later these parameters may restrict types so that they, for example, denote complete lattices. We shall therefore write  $TML[c1,c2]$  for a version of the two-level metalanguage where types are constrained as indicated by the parameters **c1** and **c2**. We then write  $\vdash_c t$  to express that the type  $t$  is well-formed with respect to the constraint **c**. Whenever we say that  $t$  is a well-formed type of  $TML[c1,c2]$  we shall mean  $\vdash_{c2} t$  because in general **c2** will be more liberal than **c1**, i.e. **c1** will imply **c2**. Next we write

$$tenv \vdash_{c1,c2} e : t$$

for the well-formedness of an expression  $e$  of intended type  $t$  assuming that the free variables of  $e$  have types as given by  $tenv$ . Here  $tenv$  is a type environment, i.e. a mapping from a finite subset of the variables  $\{x_i | i \in I\}$  to the types  $T$ . We refer to Tables 1 and 2 for the definition of  $tenv \vdash_{c1,c2} e : t$  but we point out that the constraint **c2** is used to constrain the types of variables whereas the constraint **c1** is used to constrain the types of basic expressions and combinators.

We shall say that the expression  $e$  is *closed* if it has no free variables so that  $tenv$  may be taken as a mapping from the empty set. Also we shall say that a combinator  $\psi$  is *used with type*  $t_\psi$ , if  $\vdash_{c1} t_\psi$  is the side condition that needs to be verified in order to apply the rule for  $\psi$ . As an example,  $\square$  is used with type  $(\underline{\text{Sto}} \rightarrow \text{Sto}) \rightarrow (\underline{\text{Sto}} \rightarrow \underline{\text{Sto}}) \rightarrow (\underline{\text{Sto}} \rightarrow \underline{\text{Sto}})$  in the expression for **seq** displayed above.

**Fact 3.1.4.** If  $tenv \vdash_{c1, c2} e : t_1$  and  $tenv \vdash_{c1, c2} e : t_2$  then  $t_1 = t_2$ .

### 3.1.3 Pragmatics of the metalanguage

We shall end this subsection with a few pragmatic considerations about the relationship between the two-level metalanguage and the typed  $\lambda$ -calculus we took as our starting point. One of our first points was not to pay great attention to the difference between a typed  $\lambda$ -calculus and an untyped  $\lambda$ -calculus because the various algorithms for *type analysis* might be of use in transferring types into an otherwise untyped expression. In quite an analogous way we shall not pay great attention to the difference between a typed  $\lambda$ -calculus and a two-level  $\lambda$ -calculus as one can develop an algorithm for *binding time analysis* [Nielson and Nielson, 1988a] that is useful for transferring the underlining distinction into a typed expression without this distinction. Continuing this line of argument we shall not pay great attention to the difference between a two-level  $\lambda$ -calculus and the two-level metalanguage adopted in Definition 3.1.3 because one can develop a variant of bracket abstraction (called *two-level  $\lambda$ -lifting* [Nielson and Nielson, 1988b]) that will aid in transforming underlined constructs to combinator form.

The choice of combinators in the two-level metalanguage suits the  $\lambda$ -calculus well but one may regard them as nothing but glorified versions of the basic expressions  $f_i[t]$ , i.e. that for a few of the basic expressions  $f_i[t]$  we have decided to use a different syntax. This means that one could as well study combinator-like basic expressions that would be more suitable for languages like Pascal, Prolog, Occam or *action semantics*. However, we always have the  $\lambda$ -notation available and we would only wish to restrict this in settings where the resulting metalanguage is so big as to make it hard to develop an analysis. We shall see examples of this in the next subsection where we define the parameterized semantics of the metalanguage.

## 3.2 Specification of analyses

Following most approaches to denotational semantics we shall interpret the types of the metalanguage as *domains*. We saw in section 2 that for abstract interpretation there is a special interest in the *complete lattices* and we shall restrict our attention to the *algebraic lattices* which are those complete lattices that are additionally domains. Roughly the idea will be to interpret the nonunderlined type constructs as domains whereas under-

lined type constructs will be interpreted as algebraic lattices when we are specifying abstract interpretations and as domains when we are specifying the standard interpretation.

To be selfcontained we shall briefly review a few concepts that have been treated at greater length in previous chapters of this Handbook.

**Definition 3.2.1.** A *chain* in a partially ordered set  $D=(D,\sqsubseteq)$  is a sequence  $(d_n)_n$  of elements indexed by the natural numbers such that  $d_n \sqsubseteq d_m$  whenever  $n \leq m$ . A *cpo*  $D$  is a partially ordered set with a least element,  $\perp$ , and in which every chain  $(d_n)_n$  has a (necessarily unique) least upper bound,  $\bigsqcup_n d_n$ . The cpo  $D$  is consistently complete if every subset  $Y$  of  $D$  that has an upper bound in  $D$  also has a least upper bound  $\bigsqcup Y$  in  $D$ . An element  $b$  in a cpo  $D$  is *compact* if whenever  $b \sqsubseteq \bigsqcup_n d_n$  for a chain  $(d_n)_n$  we have some natural number  $n$  such that  $b \sqsubseteq d_n$ . A subset  $B$  of  $D$  is a *basis* if every element  $d$  of  $D$  can be written as  $d = \bigsqcup_n b_n$  where  $(b_n)_n$  is a chain in  $D$  with each  $b_n$  an element of  $B$ . A *domain* is a consistently complete cpo with a countable basis  $B_D$  of compact elements. We shall use the term *algebraic lattice* for those complete lattices that are also domains, i.e. for those domains in which any subset has an upper bound.

**Definition 3.2.2.** A function  $f:D \rightarrow E$  from a domain  $D=(D,\sqsubseteq)$  to another domain  $E=(E,\sqsubseteq)$  is *monotonic* if it preserves the partial order and is *continuous* if it preserves the least upper bounds of chains, i.e.  $f(\bigsqcup_n d_n) = \bigsqcup_n f(d_n)$ . It is *additive* (sometimes called linear) if it preserves all least upper bounds, i.e.  $f(\bigsqcup Y) = \bigsqcup \{f(y) | y \in Y\}$  whenever  $Y$  has a least upper bound. It is *binary additive* if  $f(d_1 \sqcup d_2) = f(d_1) \sqcup f(d_2)$  and is *strict* if it preserves the least element, i.e.  $f(\perp) = \perp$ . It is *compact preserving* if it preserves compact elements, i.e.  $f(b) \in B_E$  whenever  $b \in B_D$ . A continuous function  $f:D \rightarrow D$  from a domain  $D=(D,\sqsubseteq)$  to itself has a least fixed point given by  $\text{FIX}(f) = \bigsqcup_n f^n(\perp)$ , i.e.  $f(\text{FIX}(f)) = \text{FIX}(f)$  and whenever  $f(d) = d$  (or indeed  $f(d) \sqsubseteq d$ ) we have  $\text{FIX}(f) \sqsubseteq d$ .

**Definition 3.2.3.** A predicate  $P$  over a domain  $D=(D,\sqsubseteq)$  is a function from  $D$  to the set  $\{\text{true}, \text{false}\}$  of truth values. It is *admissible* if  $P(\perp)$  holds and if  $P(\bigsqcup_n d_n)$  holds whenever  $(d_n)_n$  is a chain such that  $P(d_n)$  holds for every element  $d_n$ . For an admissible predicate  $P$  we have the induction principle

$$\frac{P(d) \Rightarrow P(f(d))}{P(\text{FIX}(f))}$$

whenever  $f$  is continuous. In a similar way we define the notion of admissible relation since a relation between the domains  $D_1, \dots, D_n$  ( $n \geq 1$ ) is nothing but a predicate over the cartesian product  $D_1 \times \dots \times D_n$  (where the partial order is given in the usual componentwise manner).



### 3.2.1 Interpreting the types

For a type  $t$  the definition of its interpretation  $\llbracket t \rrbracket(\mathcal{I})$  is by structure on the syntax of  $t$ . As we shall see it will make use of the parameter  $\mathcal{I}$  whenever underlined constructs are encountered. Actually, the parameter  $\mathcal{I}$  may be regarded as being a pair  $(\mathcal{I}^t, \mathcal{I}^e)$  and for the definition of  $\llbracket t \rrbracket(\mathcal{I})$  it is only the  $\mathcal{I}^t$  component that will be needed.

$\llbracket A_i \rrbracket(\mathcal{I}) =$  some a priori specified domain  $A_i$   
     with  $A_{\text{bool}}$  the domain  $\{\text{true}, \text{false}, \perp\}$  of booleans  
     and  $A_{\text{int}}$  the domain  $\{\dots, -1, 0, 1, \dots, \perp\}$  of integers

$\llbracket t_1 \times t_2 \rrbracket(\mathcal{I}) = \llbracket t_1 \rrbracket(\mathcal{I}) \times \llbracket t_2 \rrbracket(\mathcal{I})$   
     where the elements are the pairs of elements  
     and the partial order is defined componentwise

$\llbracket t_1 \rightarrow t_2 \rrbracket(\mathcal{I}) = \llbracket t_1 \rrbracket(\mathcal{I}) \rightarrow \llbracket t_2 \rrbracket(\mathcal{I})$   
     where the elements are the continuous functions  
     and the partial order is  $f \sqsubseteq g$  iff  $\forall d: f(d) \sqsubseteq g(d)$

$\llbracket A_i \rrbracket(\mathcal{I}) = \mathcal{I}_i^t$

$\llbracket t_1 \times t_2 \rrbracket(\mathcal{I}) = \mathcal{I}_\times^t(\llbracket t_1 \rrbracket(\mathcal{I}), \llbracket t_2 \rrbracket(\mathcal{I}))$

$\llbracket t_1 \rightarrow t_2 \rrbracket(\mathcal{I}) = \mathcal{I}_\rightarrow^t(\llbracket t_1 \rrbracket(\mathcal{I}), \llbracket t_2 \rrbracket(\mathcal{I}))$

The demands on the parameter  $\mathcal{I}$  are expressed in:

**Definition 3.2.4.** An *interpretation*  $\mathcal{I}$  (or  $\mathcal{I}^t$ ) of types is a specification of

- a property  $\mathcal{I}_P^t = \mathbf{P}$  of domains (e.g. “is a domain” or “is an algebraic lattice”),
- for each  $i$  a domain  $\mathcal{I}_i^t$  with property  $\mathbf{P}$ ,
- operations  $\mathcal{I}_\times^t$  and  $\mathcal{I}_\rightarrow^t$  on domains with property  $\mathbf{P}$  such that the result is a domain with property  $\mathbf{P}$ .

We shall use the term *domain interpretation* for an interpretation of types where the property  $\mathbf{P}$  equals “is a domain” and we shall use the term *lattice interpretation* for an interpretation of types where the property  $\mathbf{P}$  equals “is an algebraic lattice”.

Clearly domain interpretations are of relevance when specifying a standard semantics and lattice interpretations are of relevance when specifying abstract interpretations.

Unfortunately we will have to impose certain well-formedness conditions upon types for the above equations to define a domain. As an example,  $\text{Int} \times \underline{\text{Int}}$  will not be well-formed because

$\llbracket \text{Int} \times \underline{\text{Int}} \rrbracket(\mathcal{I}) = \mathcal{I}_\times^t(A_{\text{int}}, \mathcal{I}_{\text{int}}^t)$

and even though  $\mathcal{I}_{\text{int}}^t$  is an algebraic lattice (e.g. that for the detection of signs),  $A_{\text{int}}$  is not and so one cannot apply  $\mathcal{I}_\times^t$  when  $\mathcal{I}$  is a lattice

interpretation. Since we have argued that the use of (algebraic) lattices is a very natural setup for abstract interpretation we conclude that we should ban  $\text{Int} \times \underline{\text{Int}}$ .

With this motivation we shall define the predicates

- $\text{lt}(t)$  to ensure that  $t$  will be interpreted as an algebraic lattice when performing abstract interpretation,
- $\text{dt}(t)$  to ensure that  $t$  will be interpreted as a domain in any interpretation.

**Definition 3.2.5.** The predicates  $\text{lt}$  (for lattice type) and  $\text{dt}$  (for domain type) are defined by

	$A_i$	$\underline{A}_i$	$t_1 \times t_2$	$t_1 \underline{\times} t_2$	$t_1 \rightarrow t_2$	$t_1 \underline{\rightarrow} t_2$
$\text{lt}$	false	true	$\text{lt}_1 \wedge \text{lt}_2$	$\text{lt}_1 \wedge \text{lt}_2$	$\text{dt}_1 \wedge \text{lt}_2$	$\text{lt}_1 \wedge \text{lt}_2$
$\text{dt}$	true	true	$\text{dt}_1 \wedge \text{dt}_2$	$\text{lt}_1 \wedge \text{lt}_2$	$\text{dt}_1 \wedge \text{dt}_2$	$\text{lt}_1 \wedge \text{lt}_2$

where we write  $\text{lt}_1$  for  $\text{lt}(t_1)$  etc.

We shall regard a type  $t$  as being well-formed whenever  $\text{dt}(t)$  holds and write  $\vdash_{\text{dt}} t$  as a record of this.

**Proposition 3.2.6.** The equations for  $\llbracket t \rrbracket(\mathcal{I})$  define a domain when  $t$  is a well-formed type in  $\text{TML}[\text{dt}, \text{dt}]$  and  $\mathcal{I}$  is a domain or lattice interpretation.

**Proof.** Let  $\mathcal{I}$  be an interpretation of types that specifies the property  $\mathcal{I}_P^t = P$  where  $P(D)$  means either that  $D$  is a domain or that  $D$  is an algebraic lattice. By induction on the structure of types  $t$  we will show that

- if  $\text{dt}(t)$  then  $\llbracket t \rrbracket(\mathcal{I})$  specifies a domain,
- if  $\text{lt}(t)$  then  $\text{dt}(t)$  and  $\llbracket t \rrbracket(\mathcal{I})$  has property  $P$ .

The cases  $A_i$  and  $\underline{A}_i$  are straightforward. The case  $t_1 \times t_2$  follows because  $D_1 \times D_2$  is a domain whenever  $D_1$  and  $D_2$  are and an algebraic lattice whenever  $D_1$  and  $D_2$  are. The case  $t_1 \underline{\times} t_2$  follows from the assumptions. The case  $t_1 \rightarrow t_2$  follows because  $D_1 \rightarrow D_2$  is a domain when  $D_1$  and  $D_2$  are and an algebraic lattice when  $D_1$  is a domain and  $D_2$  is an algebraic lattice. The case  $t_1 \underline{\rightarrow} t_2$  follows from the assumptions. ■

We thus see that a type like  $\text{Int} \times \underline{\text{Int}}$  is not well-formed whereas (generalizing [Nielson, 1989]) a type like  $(\text{Int} \rightarrow \underline{\text{Int}}) \times \underline{\text{Int}}$  will be well-formed and will denote an algebraic lattice in any abstract interpretation (i.e. in any lattice interpretation). Furthermore it should now be clear that  $A_{\text{int} \times \text{bool}}$ ,  $A_{\text{int}} \times A_{\text{bool}}$ ,  $\underline{A}_{\text{int} \times \text{bool}}$ , and  $\underline{A}_{\text{int}} \times \underline{A}_{\text{bool}}$  will be treated differently in the semantics.

### 3.2.2 Interpreting the expressions

To define the meaning of a well-formed expression we shall consider a type environment  $tenv$  with domain  $\{x_1, \dots, x_n\}^2$  and a well-formed expression  $e$  of type  $t$ , i.e.

$$tenv \vdash_{dt, dt} e : t$$

Without loss of generality we may assume that  $\vdash_{dt} t_i$  whenever  $t_i = tenv(x_i)$  as otherwise  $x_i$  could be removed from the type environment (due to the formulation of the axiom for  $x_i$  in Table 1). The semantics of  $e$  relative to the interpretation  $\mathcal{I}$  is an entity

$$[e]_{tenv}(\mathcal{I}) \in [t_1](\mathcal{I}) \times \dots \times [t_n](\mathcal{I}) \rightarrow [t](\mathcal{I})$$

where again  $t_i = tenv(x_i)$ . That this makes sense is a consequence of:

**Fact 3.2.7.** If  $tenv \vdash_{dt, dt} e : t$  then  $\vdash_{dt} t$ .

The definition of  $[e]_{tenv}(\mathcal{I})$  is by structural induction on  $e$  and again we shall use the interpretation  $\mathcal{I}$ , i.e.  $(\mathcal{I}^t, \mathcal{I}^e)$ , supplied as a parameter when we come to the underlined constructs. Writing  $[e]\mathcal{I}\rho$  for  $[e]_{tenv}(\mathcal{I})(\rho)$  we have

$$\begin{aligned} [\mathbf{f}_i[t]]\mathcal{I}\rho &= \mathcal{I}_{i[t]}^e \\ [\langle e_1, e_2 \rangle]\mathcal{I}\rho &= ([e_1]\mathcal{I}\rho, [e_2]\mathcal{I}\rho) \\ [\mathbf{Tuple}\langle e_1, e_2 \rangle]\mathcal{I}\rho &= \mathcal{I}_{\mathbf{Tuple}[t]}^e([e_1]\mathcal{I}\rho)([e_2]\mathcal{I}\rho) \text{ where } \mathbf{Tuple} \text{ is used} \\ &\text{with type } t \\ [\mathbf{fst } e]\mathcal{I}\rho &= d_1 \text{ where } (d_1, d_2) = [e]\mathcal{I}\rho \\ [\mathbf{Fst } e]\mathcal{I}\rho &= \mathcal{I}_{\mathbf{Fst}[t]}^e([e]\mathcal{I}\rho) \text{ where } \mathbf{Fst} \text{ is used with type } t \\ [\mathbf{snd } e]\mathcal{I}\rho &= d_2 \text{ where } (d_1, d_2) = [e]\mathcal{I}\rho \\ [\mathbf{Snd } e]\mathcal{I}\rho &= \mathcal{I}_{\mathbf{Snd}[t]}^e([e]\mathcal{I}\rho) \text{ where } \mathbf{Snd} \text{ is used with type } t \\ [\lambda x_i[t].e]\mathcal{I}\rho &= \lambda d \in [t](\mathcal{I}). [e]_{tenv[t/x_i]}(\mathcal{I})(\rho[d/x_i]) \\ &\text{where } (d_1, \dots, d_n)[d/x_i] = (d_1, \dots, d, \dots, d_n) \text{ if } i \leq n \\ &\text{and } (d_1, \dots, d_n)[d/x_i] = (d_1, \dots, d_n, d) \text{ if } i = n+1 \\ [\mathbf{Curry } e]\mathcal{I}\rho &= \mathcal{I}_{\mathbf{Curry}[t]}^e([e]\mathcal{I}\rho) \text{ where } \mathbf{Curry} \text{ is used with type } t \\ [e_1(e_2)]\mathcal{I}\rho &= ([e_1]\mathcal{I}\rho)([e_2]\mathcal{I}\rho) \\ [\mathbf{Apply}\langle e_1, e_2 \rangle]\mathcal{I}\rho &= \mathcal{I}_{\mathbf{Apply}[t]}^e([e_1]\mathcal{I}\rho)([e_2]\mathcal{I}\rho) \text{ where } \mathbf{Apply} \text{ is used} \\ &\text{with type } t \\ [x_i]\mathcal{I}\rho &= d_i \text{ where } (d_1, \dots, d_n) = \rho \\ [\mathbf{Id}[t]]\mathcal{I}\rho &= \mathcal{I}_{\mathbf{Id}[t \rightarrow t]}^e \\ [e_1 \square e_2]\mathcal{I}\rho &= \mathcal{I}_{\square[t]}^e([e_1]\mathcal{I}\rho)([e_2]\mathcal{I}\rho) \text{ where } \square \text{ is used with type } t \end{aligned}$$

<sup>2</sup>It is rather demanding to assume that  $\text{dom}(tenv)$  is always of the form  $\{x_1, \dots, x_n\}$  for some natural number  $n$ , but as it simplifies the notation considerably we shall stick to this assumption. Alternatively, one might model a type environment as a list of pairs of the form  $(x_i, t_i)$ .

$$\begin{aligned}
\llbracket \text{Const}[t] \ e \rrbracket \mathcal{I}\rho &= \mathcal{I}_{\text{Const}[t']}^e(\llbracket e \rrbracket \mathcal{I}\rho) \text{ where } \text{Const}[t] \text{ is of type } t' \\
\llbracket \text{fix } e \rrbracket \mathcal{I}\rho &= \text{FIX}(\llbracket e \rrbracket \mathcal{I}\rho) \\
\llbracket \text{Fix } e \rrbracket \mathcal{I}\rho &= \mathcal{I}_{\text{Fix}[t]}^e(\llbracket e \rrbracket \mathcal{I}\rho) \text{ where } \text{Fix} \text{ is used with type } t \\
\llbracket \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \rrbracket \mathcal{I}\rho &= \begin{cases} \llbracket e_2 \rrbracket \mathcal{I}\rho & \text{if } \llbracket e_1 \rrbracket \mathcal{I}\rho = \text{true} \\ \llbracket e_3 \rrbracket \mathcal{I}\rho & \text{if } \llbracket e_1 \rrbracket \mathcal{I}\rho = \text{false} \\ \perp & \text{if } \llbracket e_1 \rrbracket \mathcal{I}\rho = \perp \end{cases} \\
\llbracket \text{If}\langle e_1, e_2, e_3 \rangle \rrbracket \mathcal{I}\rho &= \mathcal{I}_{\text{If}[t]}^e(\llbracket e_1 \rrbracket \mathcal{I}\rho)(\llbracket e_2 \rrbracket \mathcal{I}\rho)(\llbracket e_3 \rrbracket \mathcal{I}\rho) \text{ where } \text{If} \text{ is used} \\
&\text{with type } t
\end{aligned}$$

To prevent any misconception we point out that the pattern matching, e.g.

$$d_1 \text{ where } (d_1, d_2) = \llbracket e \rrbracket \rho \mathcal{I}$$

may be replaced by the use of explicit destructors, e.g.

$$p \downarrow 1 \text{ where } p = \llbracket e \rrbracket \rho \mathcal{I}$$

and that similarly the “where” may be replaced by textual substitution, e.g.

$$(\llbracket e \rrbracket \rho \mathcal{I}) \downarrow 1$$

The demands on the parameter  $\mathcal{I}$  are clarified by:

**Definition 3.2.8.** An interpretation  $\mathcal{I}$  is a specification of

- an interpretation  $\mathcal{I}^t$  of types that is a domain interpretation or a lattice interpretation,
- for each basic expression or combinator an entity in the required domain, i.e.

$$\mathcal{I}_{i[t]}^e \in \llbracket t \rrbracket(\mathcal{I})$$

$$\mathcal{I}_{\text{Tuple}[(t \multimap t') \rightarrow (t \multimap t'') \rightarrow (t \multimap t' \times t'')]}^e \in \llbracket t \multimap t' \rrbracket(\mathcal{I}) \rightarrow \llbracket t \multimap t'' \rrbracket(\mathcal{I}) \rightarrow \llbracket t \multimap t' \times t'' \rrbracket(\mathcal{I})$$

$$\mathcal{I}_{\text{Fst}[(t' \times t'') \rightarrow (t \multimap t')]}^e \in \llbracket t \multimap t' \times t'' \rrbracket(\mathcal{I}) \rightarrow \llbracket t \multimap t' \rrbracket(\mathcal{I})$$

$$\mathcal{I}_{\text{Snd}[(t' \times t'') \rightarrow (t \multimap t')]}^e \in \llbracket t \multimap t' \times t'' \rrbracket(\mathcal{I}) \rightarrow \llbracket t \multimap t'' \rrbracket(\mathcal{I})$$

$$\mathcal{I}_{\text{Curry}[(t' \times t'' \multimap t) \rightarrow (t' \multimap t'' \multimap t)]}^e \in \llbracket t' \times t'' \multimap t \rrbracket(\mathcal{I}) \rightarrow \llbracket t' \multimap t'' \multimap t \rrbracket(\mathcal{I})$$

$$\mathcal{I}_{\text{Apply}[(t \multimap t' \multimap t'') \rightarrow (t \multimap t') \rightarrow (t \multimap t'')]}^e \in \llbracket t \multimap t' \multimap t'' \rrbracket(\mathcal{I}) \rightarrow \llbracket t \multimap t' \rrbracket(\mathcal{I}) \rightarrow \llbracket t \multimap t'' \rrbracket(\mathcal{I})$$

$$\mathcal{I}_{\text{Id}[t \multimap t]}^e \in \llbracket t \multimap t \rrbracket(\mathcal{I})$$

$$\mathcal{I}_{\Box[(t' \multimap t'') \rightarrow (t \multimap t') \rightarrow (t \multimap t'')]}^e \in \llbracket t' \multimap t'' \rrbracket(\mathcal{I}) \rightarrow \llbracket t \multimap t' \rrbracket(\mathcal{I}) \rightarrow \llbracket t \multimap t'' \rrbracket(\mathcal{I})$$

$$\mathcal{I}_{\text{Const}[t' \rightarrow t \multimap t']}^e \in \llbracket t' \rrbracket(\mathcal{I}) \rightarrow \llbracket t \multimap t' \rrbracket(\mathcal{I})$$

$$\mathcal{I}_{\text{Fix}[(t \multimap t' \multimap t') \rightarrow (t \multimap t')]}^e \in \llbracket t \multimap t' \multimap t' \rrbracket(\mathcal{I}) \rightarrow \llbracket t \multimap t' \rrbracket(\mathcal{I})$$

$$\mathcal{I}_{\text{If}[(t \multimap \text{Bool}) \rightarrow (t \multimap t') \rightarrow (t \multimap t') \rightarrow (t \multimap t')]}^e \in \llbracket t \multimap \text{Bool} \rrbracket(\mathcal{I}) \rightarrow \llbracket t \multimap t' \rrbracket(\mathcal{I}) \rightarrow \llbracket t \multimap t' \rrbracket(\mathcal{I}) \rightarrow \llbracket t \multimap t' \rrbracket(\mathcal{I})$$

Here we must assume that the types  $t$  indexing each  $\mathcal{I}_{\psi[t]}^e$  are such that  $\llbracket \cdot \rrbracket(\mathcal{I})$  is only applied to well-formed types, i.e. types  $t'$  such

that  $\vdash_{dt} t'$ , and this is equivalent to assuming that  $\vdash_{dt} t$ .

To simplify the notation we shall henceforth feel free to omit the type and type environments as subscripts and thus write  $\llbracket e \rrbracket(I)$  for  $\llbracket e \rrbracket_{\text{tenv}}(I)$  and  $\mathcal{I}_{\psi}^e$  for  $\mathcal{I}_{\psi[t]}^e$ . (In a sense we regard the combinators as having a kind of polymorphic interpretation.)

**Proposition 3.2.9.** *The equations for  $\llbracket e \rrbracket(I)$  define a value when  $e$  is a well-formed expression in  $\text{TML}[\text{dt}, \text{dt}]$  and  $I$  is a domain or lattice interpretation.*

**Proof.** The assumptions on  $I$  ensure that  $\llbracket t \rrbracket(I)$  is a well-defined domain whenever  $\vdash_{dt} t$ . We shall show by structural induction on an expression  $e$  that

if  $\text{tenv} \vdash_{dt, dt} e : t$  where  $\text{dom}(\text{tenv}) = \{x_1, \dots, x_n\}$ ,  $\text{tenv}(x_i) = t_i$  and  $\vdash_{dt} t_i$   
 then  $\llbracket e \rrbracket_{\text{tenv}}(I) \in \llbracket t_1 \rrbracket(I) \times \dots \times \llbracket t_n \rrbracket(I) \rightarrow \llbracket t \rrbracket(I)$  and this domain does exist.

The proof makes use of Fact 3.1.4 to ensure that the type  $t$  of  $e$  is unique so that also the types indexing each  $\mathcal{I}_{\psi}^e$   $\mathcal{I}_{\psi}^e$  are unique. Furthermore it makes use of Fact 3.2.7 to ensure that the type  $t$  of  $e$  is well-formed so that, given the inference rules of Tables 1 and 2, we only request a  $\mathcal{I}_{\psi[t]}^e$  in a domain that must exist by Proposition 3.2.6. The structural induction is now mostly straightforward and we shall omit the details. ■

### 3.2.3 Example interpretations

We now present a total of five examples: a lazy standard semantics, detection of signs (in an independent attribute formulation), strictness, liveness, and detection of signs (in a relational formulation). The main point of these examples is to demonstrate the generality obtained by varying the interpretation of the underlined types and type constructors. The last two examples are somewhat technical and the details are not vital for the remainder of the development.

**Example 3.2.10 (Lazy standard semantics).** In this example we define the standard semantics of the metalanguage. This amounts to specifying a domain interpretation  $S$  and for types we have:

- the property  $S_P^t$  equals “is a domain”,
- $S_i^t = A_i$  (the a priori chosen domains for the types  $A_i$ ),
- $S_{\times}^t = \times$  (cartesian product) and  $S_{\rightarrow}^t = \rightarrow$  (continuous function space).

In other words we do not distinguish between underlined and nonunderlined types and constructors and this should not be surprising in a *standard semantics*. (We may note from this example that well-formedness of a type  $t$  is a sufficient condition for  $\llbracket t \rrbracket(S)$  to be defined but it is not necessary



as  $\llbracket t \rrbracket(\mathbf{S})$  is in fact defined as a domain for all types  $t$ .) If we wanted an *eager* standard semantics instead we might take  $\mathbf{S}_x^t$  to be a so-called *smash* product and  $\mathbf{S}_\rightarrow^t$  to be *strict* function space.

Turning to the expression part we have

$\mathbf{S}_{i[t]}^e$  is some a priori fixed element of  $\llbracket t \rrbracket(\mathbf{S})$

e.g.  $\mathbf{S}_{\text{true}}^e = \text{true}$  etc.

$\mathbf{S}_{\text{Tuple}}^e = \lambda v_1. \lambda v_2. \lambda w. (v_1(w), v_2(w))$

$\mathbf{S}_{\text{Fst}}^e = \lambda v. \lambda w. d_1$  where  $(d_1, d_2) = v(w)$

$\mathbf{S}_{\text{Snd}}^e = \lambda v. \lambda w. d_2$  where  $(d_1, d_2) = v(w)$

$\mathbf{S}_{\text{Curry}}^e = \lambda v. \lambda w_1. \lambda w_2. v(w_1, w_2)$

$\mathbf{S}_{\text{Apply}}^e = \lambda v_1. \lambda v_2. \lambda w. v_1(w)(v_2(w))$

$\mathbf{S}_{\text{Id}}^e = \lambda w. w$

$\mathbf{S}_{\square}^e = \lambda v_1. \lambda v_2. \lambda w. v_1(v_2(w))$

$\mathbf{S}_{\text{Const}}^e = \lambda v. \lambda w. v$

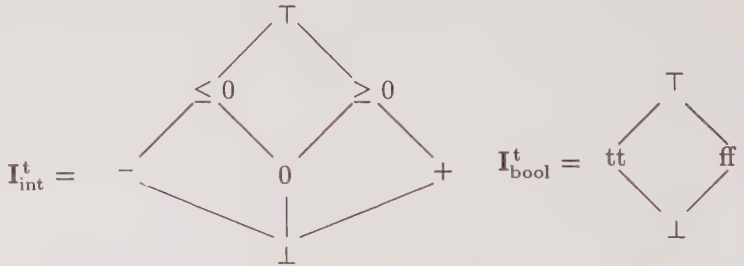
$\mathbf{S}_{\text{Fix}}^e = \lambda v. \lambda w. \text{FIX}(v(w))$

$$\mathbf{S}_{\text{If}}^e = \lambda v_1. \lambda v_2. \lambda v_3. \lambda w. \begin{cases} v_2(w) & \text{if } v_1(w) = \text{true} \\ v_3(w) & \text{if } v_1(w) = \text{false} \\ \perp & \text{if } v_1(w) = \perp \end{cases}$$

This definition is in agreement with the informal explanation of the combinators that we gave in section 3.1.

**Example 3.2.11 (Detection of signs—I).** In this example we do need the distinction between underlined and nonunderlined types in order to be able to formalize the abstract interpretation for detecting the signs of the integers. We specify a lattice interpretation  $\mathbf{I}$  and for types we have:

- the property  $\mathbf{I}_p^t$  equals “is an algebraic lattice”,
- the lattices  $\mathbf{I}_i^t$  include



- $\mathbf{I}_x^t = \times$  (cartesian product) and  $\mathbf{I}_\rightarrow^t = \rightarrow$  (continuous function space).

Turning to the expressions we have

$\mathbf{I}_{i[t]}^e$  is some a priori fixed element of  $\llbracket t \rrbracket(\mathbf{I})$

e.g.  $\mathbf{I}_{\text{true}}^e = \text{tt}$ ,  $\mathbf{I}_1^e = +$  etc.

$\mathbf{I}_{\text{Tuple}}^e = \lambda v_1. \lambda v_2. \lambda w. (v_1(w), v_2(w))$

$$\begin{aligned}
\mathbf{I}_{\text{Fst}}^e &= \lambda v. \lambda w. d_1 \text{ where } (d_1, d_2) = v(w) \\
\mathbf{I}_{\text{Snd}}^e &= \lambda v. \lambda w. d_2 \text{ where } (d_1, d_2) = v(w) \\
\mathbf{I}_{\text{Curry}}^e &= \lambda v. \lambda w_1. \lambda w_2. v(w_1, w_2) \\
\mathbf{I}_{\text{Apply}}^e &= \lambda v_1. \lambda v_2. \lambda w. v_1(w)(v_2(w)) \\
\mathbf{I}_{\text{Id}}^e &= \lambda w. w \\
\mathbf{I}_{\square}^e &= \lambda v_1. \lambda v_2. \lambda w. v_1(v_2(w)) \\
\mathbf{I}_{\text{Const}}^e &= \lambda v. \lambda w. v \\
\mathbf{I}_{\text{Fix}}^e &= \lambda v. \lambda w. \text{FIX}(v(w)) \\
\mathbf{I}_{\text{If}}^e &= \lambda v_1. \lambda v_2. \lambda v_3. \lambda w. \begin{cases} v_2(w) & \text{if } v_1(w) = \text{tt} \\ v_3(w) & \text{if } v_1(w) = \text{ff} \\ \perp & \text{if } v_1(w) = \perp \\ v_2(w) \sqcup v_3(w) & \text{if } v_1(w) = \top \end{cases}
\end{aligned}$$

To see that  $\mathbf{I}_{\text{If}}^e$  is well-defined we shall assume that the type in the subscript is well-formed. Then we have  $\text{lt}(t')$  so that by (the proof of) Proposition 3.2.6 the domain  $\llbracket t' \rrbracket(\mathbf{I})$  is a complete lattice. Hence the least upper bound exists and as the binary least upper bound operation is continuous,  $\mathbf{I}_{\text{If}}^e$  will be an element of the required (continuous) function space.

**Example 3.2.12 (Strictness).** Simplifying the lattices of the previous example we arrive at a strictness analysis. Since this analysis is by far the most cited analysis for lazy functional languages we briefly present its specification. As in the previous example we specify a lattice interpretation  $\mathbf{T}$  and for types we have:

- the property  $\mathbf{T}_p^t$  equals “is an algebraic lattice”,
- the lattices  $\mathbf{T}_i^t$  are

$$\mathbf{T}_i^t = \begin{array}{c} \bullet 1 \\ \downarrow \\ \bullet 0 \end{array}$$

- $\mathbf{T}_\times^t = \times$  (cartesian product) and  $\mathbf{T}_\rightarrow^t = \rightarrow$  (continuous function space).

Turning to the expressions we have

$\mathbf{I}_{i[t]}^e$  is some a priori fixed element of  $\llbracket t \rrbracket(\mathbf{T})$

e.g.  $\mathbf{T}_{\text{true}}^e = 1$ ,  $\mathbf{T}_1^e = 1$  etc.

$\mathbf{T}_{\text{Tuple}}^e = \lambda v_1. \lambda v_2. \lambda w. (v_1(w), v_2(w))$

$\mathbf{T}_{\text{Fst}}^e = \lambda v. \lambda w. d_1 \text{ where } (d_1, d_2) = v(w)$

$\mathbf{T}_{\text{Snd}}^e = \lambda v. \lambda w. d_2 \text{ where } (d_1, d_2) = v(w)$

$\mathbf{T}_{\text{Curry}}^e = \lambda v. \lambda w_1. \lambda w_2. v(w_1, w_2)$

$\mathbf{T}_{\text{Apply}}^e = \lambda v_1. \lambda v_2. \lambda w. v_1(w)(v_2(w))$

$\mathbf{T}_{\text{Id}}^e = \lambda w. w$

$\mathbf{T}_{\square}^e = \lambda v_1. \lambda v_2. \lambda w. v_1(v_2(w))$

$$\mathbf{T}_{\text{Const}}^e = \lambda v. \lambda w. v$$

$$\mathbf{T}_{\text{Fix}}^e = \lambda v. \lambda w. \text{FIX}(v(w))$$

$$\mathbf{T}_{\text{If}}^e = \lambda v_1. \lambda v_2. \lambda v_3. \lambda w. \begin{cases} \perp & \text{if } v_1(w)=0 \\ v_2(w) \sqcup v_3(w) & \text{if } v_1(w)=1 \end{cases}$$

Well-definedness of this specification follows much as in the previous example.

**Example 3.2.13 (Liveness).** In the previous two examples we used the ability to interpret the  $\mathbf{A}_i$  in a different manner than in the standard semantics. In this example we shall additionally need the ability to interpret the type constructor  $\underline{\rightarrow}$  in a different manner than in the standard semantics. The reason for this is that *liveness analysis* is a backwards analysis which means that the direction of the analysis is opposite to the flow of control. We specify the analysis by defining a lattice interpretation  $\mathbf{L}$  and for types we have:

- the property  $\mathbf{L}_p^t$  equals “is an algebraic lattice”,
- the lattices  $\mathbf{L}_i^t$  are

$$\mathbf{L}_i^t = \begin{array}{c} \bullet \text{live} \\ \vdots \\ \bullet \text{dead} \end{array}$$

- the operators are  $\mathbf{L}_\times^t = \times$  (cartesian product) and  $\mathbf{L}_\leftarrow^t = \leftarrow$ , i.e.  $\mathbf{L}_\leftarrow^t(D, E) = D \leftarrow E = E \rightarrow D$  which is the domain of continuous functions from  $E$  to  $D$ .

Intuitively, *dead* means “will never be used later in any computation”, while *live* means “may be used later in some computation”. It might be argued that the analysis should be called a “deadness” analysis because it is the property *dead* that can be trusted; however, it is common terminology to use the term “liveness” analysis. We should also point out that a backwards liveness analysis for flow chart programs has been seen before (in section 2.5.3).

Note that the backward nature of the analysis is recorded by interpreting  $\underline{\rightarrow}$  as  $\leftarrow$  just as the forward nature of an analysis is recorded by interpreting  $\underline{\rightarrow}$  as  $\rightarrow$  (as in the previous example).

Turning to expressions we shall impose additional constraints on the types that these are allowed to have. The motivation is that liveness analyses usually are developed for flow chart languages only and here we do not wish to give a more encompassing definition. Doing so is indeed a hard research problem as it seems to involve mixing forward and backward components into one analysis; hence interpreting  $\underline{\rightarrow}$  as  $\leftarrow$  is likely to be too simple-minded in the general case [Hughes, 1988; Ammann, 1994].

**Definition** The predicates **sr** and **sc** are defined by<sup>3</sup>

<sup>3</sup>These acronyms relate to previous papers by one of the authors and **sr** stands for

	$A_i$	$\underline{A}_i$	$t_1 \times t_2$	$t_1 \times \underline{t_2}$	$t_1 \rightarrow t_2$	$t_1 \rightarrow \underline{t_2}$
<b>sr</b>	false	true	false	$\text{sr}_1 \wedge \text{sr}_2$	false	false
<b>sc</b>	true	false	$\text{sc}_1 \wedge \text{sc}_2$	false	$\text{sc}_1 \wedge \text{sc}_2$	$\text{sr}_1 \wedge \text{sr}_2$

The intention with  $\text{sr}(t)$  is that  $t$  is an all-underlined product of base types and the intention with  $\text{sc}(t)$  is that  $t$  only contains underlined constructs if these constructs are parts of an all-underlined type with just one function space constructor in it. Clearly  $\text{sr}(t)$  implies  $\text{lt}(t)$  and hence  $\text{dt}(t)$ , and  $\text{sc}(t)$  implies  $\text{dt}(t)$ . We may thus restrict our attention to types that satisfy the predicate  $\text{sc}$ . For expressions this means that we do not need to interpret Curry (as  $(t' \underline{\times} t'' \rightarrow t) \rightarrow (t' \rightarrow t'' \rightarrow t)$  is no longer well-formed), Apply (as  $(t \rightarrow t' \rightarrow t'') \rightarrow (t \rightarrow t') \rightarrow (t \rightarrow t'')$  is no longer well-formed), Fix (as  $(t \rightarrow t' \rightarrow t') \rightarrow (t \rightarrow t')$  is no longer well-formed), or Const (as  $t' \rightarrow t \rightarrow t'$  is no longer well-formed). The expression part  $\mathbf{L}^e$  of an interpretation for TML[sc,sc] may thus be specified by

$\mathbf{L}_{i[t]}^e$  is some a priori fixed element of  $\llbracket t \rrbracket(\mathbf{L})$

$$\text{e.g. } \mathbf{L}_{=}^e = \lambda w. \begin{cases} (\top, \top) & \text{if } w = \text{live} \\ (\perp, \perp) & \text{if } w = \text{dead} \end{cases}$$

$$\mathbf{L}_{\text{Tuple}}^e = \lambda v_1. \lambda v_2. \lambda(w_1, w_2). v_1(w_1) \sqcup v_2(w_2)$$

$$\mathbf{L}_{\text{Fst}}^e = \lambda v. \lambda w. v(w, \perp)$$

$$\mathbf{L}_{\text{Snd}}^e = \lambda v. \lambda w. v(\perp, w)$$

$$\mathbf{L}_{\text{Id}}^e = \lambda w. w$$

$$\mathbf{L}_{\square}^e = \lambda v_1. \lambda v_2. \lambda w. v_2(v_1(w))$$

$$\mathbf{L}_{\text{If}}^e = \lambda v_1. \lambda v_2. \lambda v_3. \lambda w. v_1(\text{live}) \sqcup v_2(w) \sqcup v_3(w)$$

Here we should note, in particular, that  $\mathbf{L}_{\square}^e$  uses the reverse order of composition wrt.  $\mathbf{S}_{\square}^e$  and  $\mathbf{I}_{\square}^e$ .

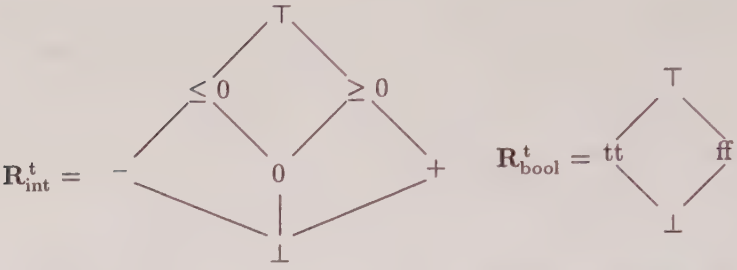
This ends Example 3.2.13.

**Example 3.2.14 (Detection of signs—II).** In our final example we shall consider an analysis where the type constructor  $\underline{\times}$  should be interpreted in a different manner than in the standard semantics. The analysis we consider is once more the detection of signs analysis but this time using a relational method where the interdependence between components in a pair is taken into account. The formalization amounts to defining a lattice interpretation  $\mathbf{R}$  but a complete treatment requires a fair amount of machinery so we refer to Nielson [1984; 1985a; 1989] and only sketch the construction. For types we have:

- the property  $\mathbf{R}_i^t$  equals “is an algebraic lattice”,
- the lattices  $\mathbf{R}_i^t$  include

---

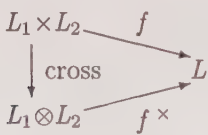
“run-time types in TML<sub>s</sub>” whereas  $\text{sc}$  stands for “compile-time types in TML<sub>s</sub>”.



- $R_{\times}^t = \otimes$  (tensor product) and  $R_{\rightarrow}^t = \rightarrow$  (continuous function space).

Note here that the relational nature of the analysis is recorded by interpreting  $\underline{\times}$  as a so-called tensor product,  $\otimes$ , just as the independent attribute nature of an analysis is recorded (in Example 3.2.11) by interpreting  $\underline{\times}$  as a cartesian product,  $\times$ . We now need to define and motivate the tensor product  $\otimes$ .

**Definition 3.2.15.** A tensor product of two algebraic lattices  $L_1$  and  $L_2$  is an algebraic lattice  $L_1 \otimes L_2$  and a separately binary additive<sup>4</sup> and continuous function  $\text{cross} : L_1 \times L_2 \rightarrow L_1 \otimes L_2$  that has the following universal property: whenever  $f : L_1 \times L_2 \rightarrow L$  is a separately binary additive and continuous function between algebraic lattices then there exists precisely one continuous and binary additive function  $f^\times : L_1 \otimes L_2 \rightarrow L$  (called the extension of  $f$ ) such that



commutes, i.e. such that  $f^\times \circ \text{cross} = f$ .

**Proposition 3.2.16.** A tensor product always exists (and it is unique to within isomorphism).

**Proof.** See [Bandelt, 1980] (or [Nielson, 1984] for an elementary proof). It is important for this result that some lattice structure is assumed as the tensor product does not exist for arbitrary domains. ■

Having been assured of the existence of the tensor product the next task is to motivate why it is relevant. We do so by calculating the tensor product in a special case and by showing that the tensor product has the ability to express the interdependence between components in a pair. For this let  $S$  be a set and  $S_\perp$  the domain with elements  $S \cup \{\perp\}$  and the partial order  $\sqsubseteq$

<sup>4</sup>A function  $f : L_1 \times L_2 \rightarrow L$  is separately binary additive if  $\lambda l_1.f(l_1, l'_2)$  and  $\lambda l_2.f(l'_1, l_2)$  are binary additive for all  $l'_1 \in L_1$  and  $l'_2 \in L_2$ .



given by  $s_1 \sqsubseteq s_2$  iff  $s_1 = s_2$  or  $s_1 = \perp$ . For a domain  $D$  in which all elements are compact, as holds for  $S_\perp$  and  $S_\perp \times S_\perp$ , the lower power domain  $\mathcal{P}_1(D)$  may be defined as

$$(\{Y \subseteq D \mid \perp \in Y \wedge \forall d \in D: \forall y \in Y: d \sqsubseteq y \Rightarrow d \in Y\}, \subseteq)$$

This is an algebraic lattice and  $\mathcal{P}_1(S_\perp)$  is isomorphic to the power set  $\mathcal{P}(S)$ . One can verify that setting  $\mathcal{P}_1(S_\perp) \otimes \mathcal{P}_1(S_\perp) = \mathcal{P}_1(S_\perp \times S_\perp)$  and  $\text{cross} = \lambda(Y_1, Y_2). Y_1 \times Y_2$  satisfies the definition of a tensor product and that the extension of a function  $f$  is given by

$$f^\times = \lambda Y. \bigsqcup \{ f(\{d_1 \mid d_1 \sqsubseteq y_1\}, \{d_2 \mid d_2 \sqsubseteq y_2\}) \mid (y_1, y_2) \in Y \}$$

This shows that in the particular case of a tensor product of power domains, the tensor product has the ability to express the interdependence between components in a pair.

**Remark** Another kind of motivation amounts to explaining the role of binary additive functions. In general an algebraic lattice imposes certain limitations upon the combinations of properties that can be expressed, for example that one cannot express the property “ $l_1$  and  $l_2$  but not  $l$ ”. (In the lattice  $\mathbf{R}_{\text{int}}^t$  for the detection of signs one may take  $l_1 = -$ ,  $l = 0$  and  $l_2 = +$ .) The binary additive functions are those functions that somehow respect these limitations. The constraining factor in the definition of  $L_1 \otimes L_2$  then is that when considering each component (as is evidenced by the demands on  $f$ ) one should respect the limitations inherent in the  $L_i$ . This means that, for example,  $\text{cross}(+, -) \sqcup \text{cross}(-, -)$  must also describe  $\text{cross}(0, -)$ .

We have to refer to Nielson [1984; 1985a] for a further discussion of the role of tensor products. For completeness we shall also finish by sketching the expression part of the lattice interpretation  $\mathbf{R}$ :

$\mathbf{R}_{[t]}^e$  is some a priori fixed element of  $\llbracket t \rrbracket(\mathbf{R})$

e.g.  $\mathbf{R}_{\text{true}}^e = \text{tt}$ ,  $\mathbf{R}_1^e = +$  etc.

$\mathbf{R}_{\text{tuple}}^e = \lambda v_1. \lambda v_2. \lambda w. \text{cross}(v_1(w), v_2(w))$

$\mathbf{R}_{\text{fst}}^e = \lambda v. \lambda w. d_1$  where  $(d_1, d_2) = \text{id}^\times(v(w))$

$\mathbf{R}_{\text{snd}}^e = \lambda v. \lambda w. d_2$  where  $(d_1, d_2) = \text{id}^\times(v(w))$

$\mathbf{R}_{\text{Curry}}^e = \lambda v. \lambda w_1. \lambda w_2. v(\text{cross}(w_1, w_2))$

$\mathbf{R}_{\text{Apply}}^e = \lambda v_1. \lambda v_2. \lambda w. v_1(w)(v_2(w))$

$\mathbf{R}_{\text{id}}^e = \lambda w. w$

$\mathbf{R}_{\square}^e = \lambda v_1. \lambda v_2. \lambda w. v_1(v_2(w))$

$\mathbf{R}_{\text{Const}}^e = \lambda v. \lambda w. v$

$\mathbf{R}_{\text{Fix}}^e = \lambda v. \lambda w. \text{FIX}(v(w))$

$$R_{\text{If}}^e = \lambda v_1. \lambda v_2. \lambda v_3. \lambda w. \begin{cases} v_2(w) & \text{if } v_1(w) = \text{tt} \\ v_3(w) & \text{if } v_1(w) = \text{ff} \\ \perp & \text{if } v_1(w) = \perp \\ v_2(w) \sqcup v_3(w) & \text{if } v_1(w) = \top \end{cases}$$

However, we should point out that with respect to the treatment given in Nielson [1984; 1989] the equations for `Tuple` and `If` are correct but too imprecise. To improve this we would need to break an argument into its *atoms* (these are the elements immediately above  $\perp$ ) and process these separately.

This ends Example 3.2.14.

### 3.2.4 Summary

To summarize, we have seen that for the purposes of abstract interpretation we need the ability to interpret underlined base types in different ways in order to describe the properties used in the different analyses. Furthermore, we have seen that the well-known distinction between forward, and backwards analyses may be formalized by the way  $\rightarrow$  is interpreted (!) and that the well-known distinction between independent attribute and relational methods may be formalized by the way  $\times$  is interpreted (!). This gives credit to the claim that a two-level metalanguage is a natural setting in which to develop a theory of abstract interpretation.

## 3.3 Correctness of analyses

To have faith in an analysis one must be able to prove that the properties resulting from the analysis are correct, e.g. with respect to the values that the standard semantics operates on. First of all this necessitates a framework in which one can *formulate* the desired correctness relations. Secondly it is desirable that the correctness follows for all terms in the metalanguage (hence all programs considered in Figure 1) once the correctness of the basic expressions and combinators has been established. (In the terminology of section 2.8 this amounts to showing that local correctness is a sufficient condition for global correctness.) Then one can consider the analyses one by one and complete the definition of the correctness relations and use this to prove the correctness of the basic expressions and combinators.

To *formulate* the correctness relations we shall adopt the framework of logical relations [Plotkin, 1980] (essentially called relational functors in [Reynolds, 1974]). For this we shall assume the existence of two domain or lattice interpretations  $\mathcal{I}$  and  $\mathcal{J}$  and the task is to define an admissible relation  $\mathcal{R}[[t]]$  between  $[[t]](\mathcal{I})$  and  $[[t]](\mathcal{J})$ , i.e.

$$\mathcal{R}[[t]] : [[t]](\mathcal{I}) \times [[t]](\mathcal{J}) \rightarrow \{\text{true}, \text{false}\}$$

in such a way that  $\mathcal{R}[[t]]$  formalizes our intuitions about correctness. We shall feel free to write  $d \mathcal{R}[[t]] e$  as well as  $\mathcal{R}[[t]](d, e)$ . In the interest of readability we prefer the notation  $\mathcal{R}[[t]]$  for  $[[t]](\mathcal{R})$  but regardless of this  $\mathcal{R}$

should be considered a parameter to  $\llbracket t \rrbracket(\dots)$ . The definition of  $\mathcal{R}[\llbracket t \rrbracket]$  is by induction on the structure of  $t$ :

$$\begin{aligned}\mathcal{R}[\mathbf{A}_i](d, e) &\equiv d = e \\ \mathcal{R}[\llbracket t_1 \times t_2 \rrbracket](d_1, d_2, e_1, e_2) &\equiv \mathcal{R}[\llbracket t_1 \rrbracket](d_1, e_1) \wedge \mathcal{R}[\llbracket t_2 \rrbracket](d_2, e_2) \\ \mathcal{R}[\llbracket t_1 \rightarrow t_2 \rrbracket](f, g) &\equiv \forall d, e: \mathcal{R}[\llbracket t_1 \rrbracket](d, e) \Rightarrow \mathcal{R}[\llbracket t_2 \rrbracket](f(d), g(e)) \\ \mathcal{R}[\mathbf{A}_i] &\equiv \mathcal{R}_i \\ \mathcal{R}[\llbracket t_1 \times t_2 \rrbracket] &\equiv \mathcal{R}_\times(\mathcal{R}[\llbracket t_1 \rrbracket], \mathcal{R}[\llbracket t_2 \rrbracket]) \\ \mathcal{R}[\llbracket t_1 \rightarrow t_2 \rrbracket](f, g) &\equiv \mathcal{R}_\rightarrow(\mathcal{R}[\llbracket t_1 \rrbracket], \mathcal{R}[\llbracket t_2 \rrbracket])\end{aligned}$$

The demands on  $\mathcal{R}$ , i.e.  $(\mathcal{R}_i)_i$ ,  $\mathcal{R}_\times$ , and  $\mathcal{R}_\rightarrow$ , are made clear in:

**Definition 3.3.1.** A *correctness correspondence* (or just *correspondence*)  $\mathcal{R}$  between domain or lattice interpretations  $\mathcal{I}$  and  $\mathcal{J}$  is a specification of

- admissible relations  $\mathcal{R}_i : \mathcal{I}_i^t \times \mathcal{J}_i^t \rightarrow \{\text{true}, \text{false}\}$ ,
- operations  $\mathcal{R}_\times$  and  $\mathcal{R}_\rightarrow$  upon admissible relations such that
 
$$\mathcal{R}_\times(\mathcal{R}_1, \mathcal{R}_2) : \mathcal{I}_\times^t(D_1, D_2) \times \mathcal{J}_\times^t(E_1, E_2) \rightarrow \{\text{true}, \text{false}\}$$

$$\mathcal{R}_\rightarrow(\mathcal{R}_1, \mathcal{R}_2) : \mathcal{I}_\rightarrow^t(D_1, D_2) \times \mathcal{J}_\rightarrow^t(E_1, E_2) \rightarrow \{\text{true}, \text{false}\}$$
 are admissible relations whenever  $\mathcal{R}_i : D_i \times E_i \rightarrow \{\text{true}, \text{false}\}$  are admissible relations,  $D_1$  and  $D_2$  are domains that satisfy the property  $\mathcal{I}_P^t$ , and  $E_1$  and  $E_2$  are domains that satisfy the property  $\mathcal{J}_P^t$ .

**Proposition 3.3.2.** The equations for  $\mathcal{R}[\llbracket t \rrbracket]$  define an admissible relation when  $t$  is a well-formed type in  $\text{TML}[\text{dt}, \text{dt}]$  and  $\mathcal{R}$  is a correctness correspondence as above.

**Proof.** We must show by structural induction on  $t$  that  $\mathcal{R}[\llbracket t \rrbracket]$  is an admissible relation between  $\llbracket t \rrbracket(\mathcal{I})$  and  $\llbracket t \rrbracket(\mathcal{J})$  and that these domains exist. This is a straightforward structural induction and we omit the details. (Note that the only reason for demanding  $t$  to be a well-formed type in  $\text{TML}[\text{dt}, \text{dt}]$ , i.e.  $\vdash_{\text{dt}} t$ , is for  $\llbracket t \rrbracket(\mathcal{I})$  and  $\llbracket t \rrbracket(\mathcal{J})$  to be guaranteed to exist.) ■

The *correctness of the basic expressions and combinators* amounts to showing that the appropriate correctness relations hold between their interpretations as given by  $\mathcal{I}$  and  $\mathcal{J}$ . We shall write  $\mathcal{R}(\mathcal{I}, \mathcal{J})$ , or  $\mathcal{I} \mathcal{R} \mathcal{J}$ , for this and the formal definition is:

whenever  $\psi$  is a basic expression or combinator and the following hold

$$\begin{aligned}\mathcal{I}_\psi^e &\in \llbracket t_1 \rrbracket(\mathcal{I}) \rightarrow \dots \llbracket t_n \rrbracket(\mathcal{I}) \rightarrow \llbracket t \rrbracket(\mathcal{I}) \\ \mathcal{J}_\psi^e &\in \llbracket t_1 \rrbracket(\mathcal{J}) \rightarrow \dots \llbracket t_n \rrbracket(\mathcal{J}) \rightarrow \llbracket t \rrbracket(\mathcal{J})\end{aligned}$$

for well-formed types  $t_1, \dots, t_n$  and  $t$  in  $\text{TML}[\text{dt}, \text{dt}]$ , then we have

$$\begin{aligned}\mathcal{R}[\llbracket t \rrbracket](d_1, e_1) \wedge \dots \wedge \mathcal{R}[\llbracket t_n \rrbracket](d_n, e_n) \\ \Rightarrow \mathcal{R}[\llbracket t \rrbracket](\mathcal{I}_\psi^e(d_1) \dots (d_n), \mathcal{J}_\psi^e(e_1) \dots (e_n))\end{aligned}$$

A shorter statement of the desired relation between  $\mathcal{I}_\psi^e$  and  $\mathcal{J}_\psi^e$  is that  $\mathcal{R}[[t_1 \rightarrow \dots t_n \rightarrow t]](\mathcal{I}_\psi^e, \mathcal{J}_\psi^e)$  must hold. This exploits the fact that  $t_1 \rightarrow \dots t_n t$  is well-formed (i.e. satisfies **dt**) if and only if all of  $t_1, \dots, t_n$  and  $t$  are and we may thus regard  $\mathcal{I}_\psi^e$  as an element of  $[[t_1 \rightarrow \dots t_n \rightarrow t]](\mathcal{I})$  and similarly for  $\mathcal{J}_\psi^e$ .

It now follows that the correctness of an analysis amounts to the correctness of the basic expressions and combinators:

**Proposition 3.3.3.** *To show the correctness  $\mathcal{R}([e](\mathcal{I})[e](\mathcal{J}))$  of a closed expression  $e$  in TML[**dt**, **dt**] it suffices to prove  $R(\mathcal{I}, \mathcal{J})$ .*

**Proof.** Let  $\mathcal{R}$  be a correctness correspondence between the domain or lattice interpretations  $\mathcal{I}$  and  $\mathcal{J}$  and such that  $\mathcal{I} \mathcal{R} \mathcal{J}$  holds. We then prove by structural induction on a well-formed expression  $e$  that

if  $\text{tenv} \vdash_{dt, dt} e : t$  with  $\text{dom}(\text{tenv}) = \{x_1, \dots, x_n\}$ ,  $\text{tenv}(x_i) = t_i$ ,  
and  $\vdash_{dt} t_i$   
then  $\mathcal{R}[[t_1]](d_1, e_1) \wedge \dots \wedge \mathcal{R}[[t_n]](d_n, e_n) \Rightarrow \mathcal{R}[[t]]([e](\mathcal{I})(d_1, \dots, d_n), [e](\mathcal{J})(e_1, \dots, e_n))$

The structural induction is mostly straightforward. In the case where  $e = \text{fix } e_0$  we use the induction principle of Definition 3.2.3. ■

**Example 3.3.4.** We shall now use the above development to show the correctness of the detection of signs analysis of Example 3.2.11 with respect to the lazy standard semantics of Example 3.2.10. The first task is to define a correctness correspondence  $\text{cor}$  between the domain interpretation **S** and the lattice interpretation **I**. We have:

- the admissible relations  $\text{cor}_i$  include

$$\text{cor}_{\text{int}}(d, p) \equiv p \sqsupseteq \begin{cases} - & \text{if } d < 0 \wedge d \neq \perp \\ 0 & \text{if } d = 0 \\ + & \text{if } d > 0 \wedge d \neq \perp \\ \perp & \text{otherwise} \end{cases}$$

$$\text{cor}_{\text{bool}}(d, p) \equiv p \sqsupseteq \begin{cases} \text{tt} & \text{if } d = \text{true} \\ \text{ff} & \text{if } d = \text{false} \\ \perp & \text{otherwise} \end{cases}$$

so that, for example,  $\text{cor}_{\text{int}}(7, \neg)$  and  $\text{cor}_{\text{bool}}(\text{true}, \top)$ ,

- the operations upon admissible relations are

$$\begin{aligned} \text{cor}_{\times}(\mathbf{R}_1, \mathbf{R}_2) ((d_1, d_2), (p_1, p_2)) &\equiv \mathbf{R}_1(d_1, p_1) \wedge \mathbf{R}_2(d_2, p_2) \\ \text{cor}_{\rightarrow}(\mathbf{R}_1, \mathbf{R}_2) (f, h) &\equiv \forall d, p: \mathbf{R}_1(d, p) \Rightarrow \mathbf{R}_2(f(d), h(p)) \end{aligned}$$

(much as for  $[[\dots \times \dots]](\mathcal{R})$  and  $[[\dots \rightarrow \dots]](\mathcal{R})$  above),

and it is straightforward to verify that this does specify a correctness correspondence.

The next task is to show  $\text{cor}(\mathbf{S}, \mathbf{I})$  so that Proposition 3.3.3 can be invoked. For the basic expressions  $\mathbf{f}_i[t]$  we must show



$\text{cor}[[t]](\mathbf{S}_i^e, \mathbf{I}_i^e)$

Little can be said here as we have not mentioned many  $f_i[t]$  in Examples 3.2.10 and 3.2.11 but we may note that

$\text{cor}[[\text{Bool}]](\text{true}, \text{tt})$

$\text{cor}[[\text{Int}]](1, +)$

both hold. For the combinators **Tuple**, **Fst**, and **Snd** related to product it is straightforward to verify the required relations as the definition of these combinators is “the same” in **S** and **I**. A similar remark holds for the combinators **Curry**, **Apply**, **Id**,  $\square$ , and **Const** related to function space. For the combinator **Fix** we may assume

$$R(ws_1, wi_1) \wedge R'(ws_2, wi_2) \Rightarrow R'(vs(ws_1)(ws_2), vi(wi_1)(wi_2))$$

$$R'(ws, wi)$$

and must show

$$R'(\text{FIX}(vs(ws)), \text{FIX}(vi(wi)))$$

where  $R$  is  $\text{cor}[[t]]$  and  $R'$  is  $\text{cor}[[t']]$  for types  $t$  and  $t'$  that both satisfy the predicate **It**. As in the proof of Proposition 3.3.3 this follows using the induction principle of Definition 3.2.3. Finally for the combinator **If** we may assume

$$R(ws, wi) \Rightarrow vi_1(wi) \sqsupseteq \begin{cases} \text{tt} & \text{if } vs_1(ws) = \text{true} \\ \text{ff} & \text{if } vs_1(ws) = \text{false} \\ \perp & \text{otherwise} \end{cases}$$

$$R(ws, wi) \Rightarrow R'(vs_2(ws), vi_2(wi))$$

$$R(ws, wi) \Rightarrow R'(vs_3(ws), vi_3(wi))$$

$$R(ws, wi)$$

and must show

$$R' \left( \begin{cases} vs_2(ws) & \text{if } vs_1(ws) = \text{true} \\ vs_3(ws) & \text{if } vs_1(ws) = \text{false} \\ \perp & \text{otherwise} \end{cases} \right), \left( \begin{cases} vi_2(wi) & \text{if } vi_1(wi) = \text{tt} \\ vi_3(wi) & \text{if } vi_1(wi) = \text{ff} \\ vi_2(wi) \sqcup vi_3(wi) & \text{if } vi_1(wi) = \top \\ \perp & \text{otherwise} \end{cases} \right)$$

where again  $R$  is  $\text{cor}[[t]]$  and  $R'$  is  $\text{cor}[[t']]$  for types  $t$  and  $t'$  that both satisfy the predicate **It**. The proof amounts to considering each of the cases  $vs_1(ws) = \text{true}$ ,  $vs_1(ws) = \text{false}$ , and  $vs_1(ws) = \perp$  separately and will need:

**Fact** If  $t$  satisfies **It** then  $\text{cor}[[t]](ws, wi) \wedge wi \sqsubseteq wi'$  implies  $\text{cor}[[t]](ws, wi')$ .

The proof of this fact is by structural induction on  $t$ . The case  $t = \underline{A}_i$  can only be conducted if we tacitly assume that  $\underline{A}_i$  is one of  $\underline{A}_{\text{bool}}$  or  $\underline{A}_{\text{int}}$ .

For reasons of space we shall not prove the correctness of the remaining analyses defined in section 3.2. There are no profound difficulties in establishing the correctness of the detection of signs analysis defined in Example 3.2.14. For the liveness analysis of Example 3.2.13 the notion of a correctness correspondence is too weak but a variation of the development



presented here may be used to prove its correctness (see [Nielson, 1989]). We should point out that the complications in the proof of correctness of the liveness analysis are due to the fact that the properties in the liveness analysis do not describe actual values but rather their subsequent use in future computations. The terms *first-order* analyses (e.g. detection of signs) and *second-order* analyses (e.g. liveness) have been used for this distinction by Nielson [1985b; 1989].

### 3.3.1 Safety: comparing two analyses

A special case of correctness is when one compares two analyses and shows that the properties resulting from one analysis correctly describe the properties resulting from the other. We shall use the term *safety* for this and we shall see in the next subsection that from the safety of one analysis with respect to a correct analysis one is often able to infer the correctness of the former analysis.

As an example we might consider two analyses that operate on the same properties but have different ways of modelling the basic expressions and combinators. We formalize this by considering two lattice interpretations  $\mathcal{I}$  and  $\mathcal{J}$  with  $\mathcal{I}_P^t = \mathcal{J}_P^t$ ,  $\mathcal{I}_i^t = \mathcal{J}_i^t$ ,  $\mathcal{I}_\times^t = \mathcal{J}_\times^t$ , and  $\mathcal{I}_{\rightarrow}^t = \mathcal{J}_{\rightarrow}^t$  (in short  $\mathcal{I}^t = \mathcal{J}^t$ ). When we want to be more specific we shall let  $\mathcal{I}$  be the interpretation **I** for the detection of signs (Example 3.2.10). If we wish to express that the results of  $\mathcal{J}$  are coarser than those of  $\mathcal{I}$ , e.g. that  $\llbracket e \rrbracket(\mathcal{I}) = +$  whereas  $\llbracket e \rrbracket(\mathcal{J}) = \neg$ , we must define a correctness correspondence and we shall use the notation  $\leq$ . Given the motivation presented in section 2 we take

$$\begin{aligned} \leq_i &\equiv \sqsubseteq \\ \leq_\times(R_1, R_2) &\equiv \sqsubseteq \\ \leq_{\rightarrow}(R_1, R_2) &\equiv \sqsubseteq \end{aligned}$$

because the idea was to use the partial order  $\sqsubseteq$  to express the amount of precision among various properties<sup>5</sup>. In a more abstract way one might say that a correspondence  $\mathcal{R}$  between two lattice interpretations  $\mathcal{I}$  and  $\mathcal{J}$  is a *safety correspondence* when  $\mathcal{R}_i \equiv \sqsubseteq$ ,  $\mathcal{R}_\times(\sqsubseteq, \sqsubseteq) \equiv \sqsubseteq$ , and  $\mathcal{R}_{\rightarrow}(\sqsubseteq, \sqsubseteq) \equiv \sqsubseteq$  whenever  $\mathcal{I}^t = \mathcal{J}^t$ . Clearly  $\leq$  is a safety correspondence.

We shall claim that  $\leq$  is the proper relation to use for relating  $\mathcal{I}$  and  $\mathcal{J}$ . On an all-underlined type  $t$  (e.g. Int or Int $\rightarrow$ Int) the relation  $\leq[t]$  clearly equals  $\sqsubseteq$  which is the relation that also section 2 used. On a type  $t$  without any underlined symbols (e.g. Int or Int $\rightarrow$ Int) it is straightforward to see (as we show below) that  $\leq[t]$  equals  $=$  and this is the correct relation to use given that we only perform abstract interpretation on underlined base types and constructors. In particular,  $\leq[t]$  is more adequate than  $\sqsubseteq$  in this case.

<sup>5</sup>This need not be so in general (see [Mycroft and Nielson, 1983]) but considerably simplifies the technical development.

However, there are types upon which  $\leq[t]$  behaves in a strange way. As an example let  $t_0 = \underline{\text{Bool}} \rightarrow \text{Bool}$  and consider a basic expression  $f_0[t_0]$  such that

$$\mathcal{I}_0^e = \mathcal{J}_0^e = \lambda d. \begin{cases} \text{true} & \text{if } d = \top \\ \perp & \text{otherwise} \end{cases}$$

Then  $\mathcal{I}_0^e \leq [t_0]$   $\mathcal{J}_0^e$  fails because we have  $\perp \sqsubseteq \top$  but not  $\perp = \text{true}$ . This means that  $\leq[t_0]$  is not even reflexive. Clearly we want  $\leq[t]$  to be a partial order and it is also natural to assume that it implies  $\sqsubseteq$  because we have argued for the use of  $\sqsubseteq$  to compare properties of an all-underlined type. We shall achieve this by restricting the types to be considered just as we did to ensure that  $[t](\mathcal{I})$  and  $[t](\mathcal{J})$  were domains.

First we need a few definitions:

**Definition 3.3.5.** A suborder  $\leq$  on a domain  $D = (D, \sqsubseteq)$  is a partial order that satisfies

$$d_1 \leq d_2 \Rightarrow d_1 \sqsubseteq d_2$$

for all  $d_1$  and  $d_2$ .

For an arbitrary safety correspondence  $\mathcal{R}$ , e.g.  $\leq$ , this motivates defining the predicates

**pt**( $t$ ) to ensure that  $\mathcal{R}[t]$  amounts to  $=$ ,

**it**( $t$ ) to ensure that  $\mathcal{R}[t]$  amounts to  $\sqsubseteq$ ,

**lpt**( $t$ ) to ensure that  $\mathcal{R}[t]$  is a suborder.

The predicate **pt** was called *pure* in Nielson [1988; 1989] because it will restrict the types to have no underlined symbols. The predicates **it** and **lpt** should be thought of as slightly more discriminating analogues of **lt** and **dt**. They were called *impure* and *level-preserving*, respectively, in Nielson [1988; 1989] but with one difference:  $\text{TML}[\text{dt}, \text{dt}]$  allows more well-formed types than does [Nielson, 1988] or [Nielson, 1989].

**Definition 3.3.6.** The predicates **pt** (for *pure type*), **it** (for *impure type*), and **lpt** (for *level-preserving type*) are defined by:

	$A_i$	$\underline{A}_i$	$t_1 \times t_2$	$t_1 \underline{\times} t_2$	$t_1 \rightarrow t_2$	$t_1 \underline{\rightarrow} t_2$
<b>pt</b>	true	false	$\text{pt}_1 \wedge \text{pt}_2$	false	$\text{pt}_1 \wedge \text{pt}_2$	false
<b>it</b>	false	true	$\text{it}_1 \wedge \text{it}_2$	$\text{it}_1 \wedge \text{it}_2$	$\text{lpt}_1 \wedge \text{it}_2$	$\text{it}_1 \wedge \text{it}_2$
<b>lpt</b>	true	true	$\text{lpt}_1 \wedge \text{lpt}_2$	$\text{it}_1 \wedge \text{it}_2$	$(\text{pt}_1 \wedge \text{lpt}_2) \vee (\text{lpt}_1 \wedge \text{it}_2)$	$\text{it}_1 \wedge \text{it}_2$

Note that the difference between **lt** and **dt** versus **it** and **lpt** is due to the difference between the definition of  $\text{dt}(t_1 \rightarrow t_2)$  and  $\text{lpt}(t_1 \rightarrow t_2)$ .

**Lemma 3.3.7.** The formal definition of the predicates **pt**, **it**, and **lpt** satisfy the intentions displayed above.

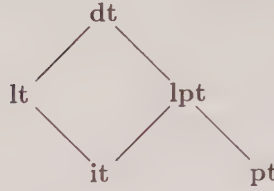


Fig. 2. Well-formedness constraints

**Proof.** For an arbitrary safety relation  $\mathcal{R}$ , e.g.  $\leq$ , between lattice interpretations  $\mathcal{I}$  and  $\mathcal{J}$  with  $\mathcal{I}^t = \mathcal{J}^t$  we prove by induction on types  $t$  that

- $\mathbf{pt}(t) \Rightarrow \mathbf{lpt}(t) \wedge \mathcal{R}[t] \equiv \equiv$ ,
- $\mathbf{it}(t) \Rightarrow \mathbf{lpt}(t) \wedge \mathbf{lt}(t) \wedge \mathcal{R}[t] \equiv \sqsubseteq$ ,
- $\mathbf{lpt}(t) \Rightarrow \mathbf{dt}(t) \wedge \mathcal{R}[t]$  is a suborder.

The first result is a straightforward structural induction and we shall not give any details. The second and third results must be proved jointly as  $\mathbf{it}$  and  $\mathbf{lpt}$  are mutually interdependent.

The cases  $\mathbf{A}_i$  and  $\mathbf{A}_i$  are straightforward. In the case  $t = t_1 \times t_2$  we first assume  $\mathbf{it}(t)$ . Then  $\mathbf{it}(t_1)$  and  $\mathbf{it}(t_2)$  so that  $\mathbf{lpt}(t_1)$ ,  $\mathbf{lpt}(t_2)$ ,  $\mathbf{lt}(t_1)$ ,  $\mathbf{lt}(t_2)$ ,  $\mathcal{R}[t_1] \equiv \sqsubseteq$ , and  $\mathcal{R}[t_2] \equiv \sqsubseteq$ . It follows that  $\mathbf{lpt}(t)$ ,  $\mathbf{lt}(t)$ , and  $\mathcal{R}[t] \equiv \sqsubseteq$ . Next we assume  $\mathbf{lpt}(t)$ . Then  $\mathbf{lpt}(t_1)$  and  $\mathbf{lpt}(t_2)$  so that  $\mathbf{dt}(t_1)$ ,  $\mathbf{dt}(t_2)$ ,  $\mathcal{R}[t_1]$  is a suborder, and  $\mathcal{R}[t_2]$  is a suborder. It follows that  $\mathbf{dt}(t)$  and that  $\mathcal{R}[t]$  is a suborder. In the case  $t = t_1 \sqcup t_2$  the assumptions  $\mathbf{it}(t)$  and  $\mathbf{lpt}(t)$  are equivalent so we assume that  $\mathbf{it}(t)$  holds. Then  $\mathbf{it}(t_1)$  and  $\mathbf{it}(t_2)$  so that  $\mathbf{lt}(t_1)$ ,  $\mathbf{lt}(t_2)$ ,  $\mathcal{R}[t_1] \equiv \sqsubseteq$ , and  $\mathcal{R}[t_2] \equiv \sqsubseteq$ . It follows that  $\mathbf{lpt}(t)$ ,  $\mathbf{lt}(t)$ ,  $\mathbf{dt}(t)$ , and  $\mathcal{R}[t] \equiv \sqsubseteq$  which is a suborder. The case  $t = t_1 \rightarrow t_2$  is similar.

In the final case  $t = t_1 \rightarrow t_2$  we first assume  $\mathbf{it}(t)$ . Then  $\mathbf{lpt}(t_1)$  and  $\mathbf{it}(t_2)$  so that  $\mathbf{dt}(t_1)$ ,  $\mathbf{lt}(t_2)$ ,  $\mathcal{R}[t_1]$  is a suborder, and  $\mathcal{R}[t_2] \equiv \sqsubseteq$ . It follows that  $\mathbf{lpt}(t)$ ,  $\mathbf{lt}(t)$ , and hence also  $\mathbf{dt}(t)$ . The relation  $\mathcal{R}[t]$  holds on  $(f, g)$  when

$$\mathcal{R}[t_1](d, e) \Rightarrow f(d) \sqsubseteq g(e)$$

for all  $d$  and  $e$ . If  $f \sqsubseteq g$  and  $\mathcal{R}[t_1](d, e)$  we have  $d \sqsubseteq e$  and hence  $f(d) \sqsubseteq g(e)$  so that  $\mathcal{R}[t](f, g)$ . If  $\mathcal{R}[t](f, g)$  we have  $\mathcal{R}[t_1](d, d)$  and hence  $f(d) \sqsubseteq g(d)$  for all  $d$  and this amounts to  $f \sqsubseteq g$ . Thus  $\mathcal{R}[t]$  equals the suborder  $\sqsubseteq$ . Next we assume  $\mathbf{lpt}(t)$ . There are two cases to consider but we have just treated  $\mathbf{lpt}(t_1) \wedge \mathbf{it}(t_2)$  so that we may assume  $\mathbf{pt}(t_1)$  and  $\mathbf{lpt}(t_2)$ . It follows that  $\mathbf{lpt}(t_1)$ ,  $\mathbf{dt}(t_1)$ ,  $\mathcal{R}[t_1] \equiv \equiv$ ,  $\mathbf{dt}(t_2)$ , and  $\mathcal{R}[t_2]$  is a suborder. Hence  $\mathbf{dt}(t)$  and the relation  $\mathcal{R}[t]$  holds on  $(f, g)$  whenever

$$\forall d: \mathcal{R}[t_2](f(d), g(d))$$

and this is clearly a suborder. ■

### 3.3.2 Summary

We shall now restrict the types of the basic expressions and combinators so that they have level-preserving types. This amounts to considering  $\text{TML}[\text{lpt}, \text{dt}]$  as there is no need also to require the types of variables to be level-preserving. For the basic expressions  $f_i[t]$  this condition simply amounts to requiring  $t$  to be level-preserving, i.e. satisfy the predicate **lpt**. For the combinators a general form of their types may be found in the side conditions in Tables 1 and 2. These general forms are expressed in terms of subtypes  $t, t', t'', t_0, t_1$ , and  $t_2$  and the restriction to  $\text{TML}[\text{lpt}, \text{dt}]$  amounts to demanding that all these subtypes are impure, i.e. satisfy the predicate **it**.

The relationship between  $\text{TML}[\text{lpt}, \text{dt}]$  and  $\text{TML}[\text{dt}, \text{dt}]$  is clarified by:

**Fact 3.3.8.** If  $\text{tenv} \vdash_{\text{pt}, \text{dt}} e : t$  then  $\text{tenv} \vdash_{\text{dt}, \text{dt}} e : t$  and hence  $\vdash_{\text{dt}} t$ .

Thus  $\text{TML}[\text{lpt}, \text{dt}]$  is a proper subset of  $\text{TML}[\text{dt}, \text{dt}]$  and Fact 3.1.4, Proposition 3.2.6, Proposition 3.2.9, Proposition 3.3.2, and Proposition 3.3.3 apply to  $\text{TML}[\text{lpt}, \text{dt}]$  as well.

Given Lemma 3.3.7 we then have that  $\leq$  is a partial order in the collection of interpretations for  $\text{TML}[\text{lpt}, \text{dt}]$  contrary to what is the case when one considers the collection of all interpretations for  $\text{TML}[\text{dt}, \text{dt}]$ . In particular we have  $\mathcal{I} \leq \mathcal{J}$  whenever  $\mathcal{I}$  is an interpretation for  $\text{TML}[\text{lpt}, \text{dt}]$  and by Proposition 3.3.3 we then have  $(\llbracket e \rrbracket(\mathcal{I})) \leq \llbracket t \rrbracket(\llbracket e \rrbracket(\mathcal{I}))$  for all closed expressions  $e$  of type  $t$  (even if  $t$  is not level-preserving).

## 3.4 Induced analyses

One shortcoming of the development of the previous subsection is that a correct analysis may be so imprecise as to be practically useless. An example is an analysis where all basic expressions and combinators are interpreted as the greatest element  $\top$  (whenever they are used with a lattice type). The notion of correctness is topological in nature but we would ideally like something that was a bit more metric in nature so that we could express *how* imprecise a correct analysis is. Unfortunately no one has been able to develop an adequate metric for these purposes.

The alternative then is to compare various analyses. We shall take the point of view that the choice of the type part of an interpretation, i.e. the choice of what properties to use for underlined types etc., represents a deliberate choice as to the degree of precision that is desired<sup>6</sup>. Thus the definition of  $\leq$  in the previous subsection allows us to compare various analyses provided that they use the same selection of properties. So if we are confronted with two analyses we may compare them and might be able to say that one analysis is more imprecise than (i.e.  $\geq$ ) another and so

<sup>6</sup>This is a more restricted point of view than is put forward in [Steffen, 1987].



we might prefer the other analysis. This is not a complete recipe as  $\leq$  is only a partial order and in general not a total order. Also even if we have preferred some analysis there is no easy way to tell whether we could develop an analysis that would be even more precise.

This motivates the development in the present subsection where we show that under certain circumstances there is a *most precise* analysis over a given selection of properties. Following [Cousot and Cousot, 1979] we shall term this the *induced* analysis. As we shall see in the next subsection there may well be pragmatic reasons for adopting an analysis that is less precise than the *induced* analysis. However, even if one does so we believe that the induced analysis serves an important role as a standard against which analyses may be compared: whenever the analysis of one's choice models a basic expression or combinator less precisely than the induced analysis does then one may judge the degree of imprecision and decide whether it is warranted for pragmatic reasons (e.g. termination, low time-complexity, easy to implement, etc.).

For the technical development we shall assume that we have a domain or lattice interpretation  $\mathcal{I}$  and a lattice interpretation  $\mathcal{J}$ . (Actually, we only need the type part of  $\mathcal{J}$  and for the majority of the development we also only need the type part of  $\mathcal{I}$ .) Here one should think of  $\mathcal{I}$  as the standard semantics, e.g. the lazy standard semantics of Example 3.2.10, and one should think of  $\mathcal{J}$  as some analysis, e.g. the detection of signs of Example 3.2.11. However, the development also specializes to the case where  $\mathcal{I}$  is some analysis much as the notion of correctness correspondence in the previous subsection specialized to the notion of safety correspondence.

We then propose to define a transformation function  $\beta[t]$  from  $\llbracket t \rrbracket(\mathcal{I})$  to  $\llbracket t \rrbracket(\mathcal{J})$ , i.e.

$$\beta[t] : \llbracket t \rrbracket(\mathcal{I}) \rightarrow \llbracket t \rrbracket(\mathcal{J}).$$

Again one should regard  $\beta$  as a parameter to  $\llbracket t \rrbracket(\dots)$  just as  $\mathcal{I}$  and  $\mathcal{J}$  are. The definition of  $\beta[t]$  is by induction on the structure of  $t$ :

$$\beta[A_i] \equiv \lambda d. d$$

$$\beta[t_1 \times t_2] \equiv \lambda (d_1, d_2). (\beta[t_1](d_1), \beta[t_2](d_2))$$

$$\beta[t_1 \rightarrow t_2] \equiv \lambda f. \lambda p. \bigsqcup \{ \beta[t_2](f(d)) \mid \beta[t_1](d_1) \sqsubseteq p \}$$

$$\beta[A_i] \equiv \beta_i$$

$$\beta[t_1 \times t_2] \equiv \beta_{\times}(\beta[t_1], \beta[t_2])$$

$$\beta[t_1 \rightarrow t_2] \equiv \beta_{\rightarrow}(\beta[t_1], \beta[t_2])$$

Several points now need to be addressed. First we must clarify the claims we shall make about the functions  $\beta[t]$  and the demands that this enforces on the parameter  $\beta$ . Secondly we must find a way of constraining the types  $t$  such that the functions  $\beta[t]$  exist and have the desired properties. Finally, we must show that the definition of  $\beta[t]$  is as intended, and in particular that it is correct. Closely related to this is the question of why the equation



for  $\beta[t_1 \rightarrow t_2]$  uses  $\sqsubseteq$  and  $\sqcup$  rather than  $\leq[\dots]$  and its associated least upper bound operator  $\bigvee$ .

### 3.4.1 Existence

First we need some definitions and simple facts:

**Definition 3.4.1.** A *representation transformation* (or just a *transformation*)  $f$  from a domain  $D=(D,\sqsubseteq)$  to a domain  $E=(E,\sqsubseteq)$  is a function that is strict, continuous, and compact preserving (see Definition 3.2.2).

**Fact 3.4.2.** Let  $(\alpha,\gamma)$  be a pair of adjointed functions between algebraic lattices. Then  $\alpha$  is strict and continuous but not necessarily compact preserving. If  $\gamma$  is additionally continuous then  $\alpha$  is compact preserving.

**Example 3.4.3.** Recall the definition of  $S_{\text{int}}^t$  and  $I_{\text{int}}^t$  in Examples 3.2.10 and 3.2.11. A representation transformation from  $S_{\text{int}}^t$  to  $I_{\text{int}}^t$  may be defined by

$$\lambda d. \begin{cases} + & \text{if } d > 0 \\ 0 & \text{if } d = 0 \\ - & \text{if } d < 0 \\ \perp & \text{if } d = \perp \end{cases}$$

(Note that all elements in  $S_{\text{int}}^t$  and  $I_{\text{int}}^t$  are compact.)

The demands on the parameter  $\beta$  are clarified by:

**Definition 3.4.4.** A *representation transformer*  $\beta$  from a domain or lattice interpretation  $\mathcal{I}$  to a lattice interpretation  $\mathcal{J}$  is a specification of:

- representation transformations  $\beta_i : \mathcal{I}_i^t \rightarrow \mathcal{J}_i^t$ ,
- operations  $\beta_{\times}$  and  $\beta_{\rightarrow}$  such that

$$\beta_{\times}(f_1, f_2) : \mathcal{I}_{\times}^t(D_1, D_2) \rightarrow \mathcal{J}_{\times}^t(E_1, E_2)$$

$$\beta_{\rightarrow}(f_1, f_2) : \mathcal{I}_{\rightarrow}^t(D_1, D_2) \rightarrow \mathcal{J}_{\rightarrow}^t(E_1, E_2)$$

are representation transformations whenever  $f_i : D_i \rightarrow E_i$  are representation transformations,  $D_1$  and  $D_2$  are domains that satisfy the property  $\mathcal{I}_P^t$ , and  $E_1$  and  $E_2$  are algebraic lattices.

**Example 3.4.5.** A representation transformer  $b$  from the interpretation  $S$  of Example 3.2.10 to the interpretation  $I$  of Example 3.2.11 may be defined by:

- representation transformations  $b_i : S_i^t \rightarrow I_i^t$

$$\text{with } b_{\text{int}} = \lambda d. \begin{cases} + & \text{if } d > 0 \\ 0 & \text{if } d = 0 \\ - & \text{if } d < 0 \\ \perp & \text{if } d = \perp \end{cases}$$

$$\text{and } b_{\text{bool}} = \lambda d. \begin{cases} \text{tt} & \text{if } d = \text{true} \\ \text{ff} & \text{if } d = \text{false} \\ \perp & \text{if } d = \perp \end{cases}$$

- operations  $b_{\times}$  and  $b_{\rightarrow}$  given by

$$\begin{aligned} b_{\times}(f_1, f_2) &= \lambda(d_1, d_2). (f_1(d_1), f_2(d_2)) \\ b_{\rightarrow}(f_1, f_2) &= \lambda f. \lambda p. \bigsqcup \{ f_2(f(d)) \mid f_1(d) \sqsubseteq p \} \end{aligned}$$

Here we cannot be more specific about the  $b_i$  as Example 3.2.11 only is specific about  $I_{\text{int}}^t$  and  $I_{\text{bool}}^t$  but clearly the  $b_{\text{int}}$  and  $b_{\text{bool}}$  exhibited are representation transformations. Assuming that  $f_1$  and  $f_2$  are representation transformations the well-definedness of  $b_{\times}(f_1, f_2)$  is immediate. It is clearly strict and continuous and it preserves compact elements because the compact elements in a cartesian product are the pairs of compact elements in each component. Also  $b_{\rightarrow}(f_1, f_2)$  is well-defined because the least upper bound is taken in an algebraic lattice (called  $E_2$  in Definition 3.4.4). That  $b_{\rightarrow}(f_1, f_2)(f)$  is a continuous function and that  $b_{\rightarrow}(f_1, f_2)$  is a representation transformation is slightly more involved. We omit the details as they follow rather easily from the case  $t=t_1 \rightarrow t_2$  in the proof of the following proposition.

Well-definedness of  $\beta[t]$  follows from the following fact and proposition:

**Fact 3.4.6.** If  $t$  is a pure type and  $\beta$  a representation transformer then  $\beta[t]$  is the representation transformation  $\lambda d. d$  and  $[t](I) = [t](J)$ .

**Proposition 3.4.7.** The equations for  $\beta[t]$  define a representation transformation when  $t$  is a level-preserving type and  $\beta$  is a representation transformer.

**Proof.** We show by structural induction on  $t$  that if  $\text{lpt}(t)$  then the above equations define a function

$$\beta[t] : [t](I) \rightarrow [t](J)$$

and that this function is a representation transformation.

The case  $t = A_i$  is straightforward. The case  $t = t_1 \times t_2$  follows from the induction hypothesis given that the compact elements in a cartesian product  $D' \times D''$  are the pairs of compact elements of  $D'$  and  $D''$  respectively, i.e.  $B_{D' \times D''} = B_{D'} \times B_{D''}$ . The case  $t = A_i$  follows from the assumptions on  $\beta$ . In a similar way the cases  $t = t_1 \times t_2$  and  $t = t_1 \rightarrow t_2$  follow from the assumptions on  $\beta$  and the induction hypothesis.

It remains to consider the case where  $t = t_1 \rightarrow t_2$ . There are two “alternatives” in the definition of  $\text{lpt}(t_1 \rightarrow t_2)$  so we first consider the possibility where  $t_1$  is pure and  $t_2$  is level-preserving. We shall write

$$Y(f, p) = \{ \beta[t_2](f(d)) \mid \beta[t_1](d) \sqsubseteq p \}$$

and by Fact 3.4.6 we get  $Y(f, p) = \{ \beta[t_2](f(d)) \mid d \sqsubseteq p \}$ . By continuity of  $f$  and  $\beta[t_2]$  this set contains an upper bound for itself, namely  $\beta[t_2](f(p))$ . Hence  $\bigsqcup Y(f, p)$  always exists and equals  $\beta[t_2](f(d))$ . Next we consider the situation where  $t_1$  is level-preserving and  $t_2$  is impure. Then  $t_2$  is also a lattice type, i.e.  $\text{lt}(t_2)$ , so that  $[t_2](J)$  is an algebraic lattice. It is then straightforward that  $\bigsqcup Y(f, p)$  exists.

We have now shown that  $\beta[t_1 \rightarrow t_2](f)(p)$  always exists when  $t_1 \rightarrow t_2$  is level-preserving. To show that  $\beta[t_1 \rightarrow t_2](f)$  exists we must show that  $\sqcup Y(f, p)$  depends continuously on  $p$ , i.e. that  $\lambda p. \sqcup Y(f, p)$  is continuous. First we write

$$Z(f, p) = \{ \beta[t_2](f(b)) \mid \beta[t_1](b) \sqsubseteq p \wedge b \text{ is compact} \}$$

This set has  $\sqcup Y(f, p)$  as an upper bound so by consistent completeness  $\sqcup Z(f, p)$  exists and we clearly have  $\sqcup Z(f, p) \sqsubseteq \sqcup Y(f, p)$ . Actually,  $\sqcup Z(f, p) = \sqcup Y(f, p)$  as any element  $d$  such that  $\beta[t_1](d) \sqsubseteq p$  may be written as  $d = \sqcup_n b_n$  where each  $b_n$  is compact. Then  $Z(f, p)$  contains all  $\beta[t_2](f(b_n))$  so  $\sqcup Z(f, p) \supseteq \sqcup_n \beta[t_2](f(b_n)) = \beta[t_2](f(d))$  and hence  $\sqcup Z(f, p) \supseteq \sqcup Y(f, p)$  as  $d$  was arbitrary.

To show that  $\lambda p. \sqcup Z(f, p)$  is continuous let  $p = \sqcup_n p_n$ . Clearly  $\sqcup_n \sqcup Z(f, p_n) \sqsubseteq \sqcup Z(f, p)$  so it suffices to show that  $\sqcup Z(f, p) \sqsubseteq \sqcup_n \sqcup Z(f, p_n)$ . If  $b$  is compact and  $\beta[t_1](b) \sqsubseteq \sqcup_n p_n$  then by the induction hypothesis  $\beta[t_1](b)$  is compact so that  $\beta[t_1](b) \sqsubseteq p_n$  for some  $n$ . Hence  $\beta[t_2](f(b)) \sqsubseteq \sqcup Z(f, p_n)$  and this shows the result.

Finally, we must show that  $\beta[t_1 \rightarrow t_2]$  is a representation transformation. So observe that  $\beta[t_1 \rightarrow t_2](\perp) = \perp$  follows because  $\beta[t_2]$  is strict. That  $\beta[t_1 \rightarrow t_2]$  is continuous follows because  $\beta[t_2]$  is. It now remains to show that  $\beta[t_1 \rightarrow t_2]$  preserves compact elements. For a domain  $D \rightarrow E$ , where also  $D$  and  $E$  are domains, we shall write

$$[d, e] = \lambda d'. \begin{cases} e & \text{if } d' \sqsupseteq d \\ \perp & \text{otherwise} \end{cases}$$

The function is continuous if  $d$  is compact and is a compact element of  $D \rightarrow E$  if additionally  $e$  is compact. The general form of a compact element in  $D \rightarrow E$  is

$$[b_1, e_1] \sqcup \dots \sqcup [b_n, e_n]$$

(for  $n \geq 1$ ) where all  $b_i$  and  $e_i$  are compact and we assume that  $\{e_j \mid j \in J_{d'}\}$  has an upper bound (and by consistent completeness a least upper bound) whenever  $d' \in D$  and  $J_{d'} = \{j \mid b_j \sqsubseteq d'\}$ . We shall say that  $[b_1, e_1] \sqcup \dots \sqcup [b_n, e_n]$  is a *complete listing* if for all  $d' \in D$  there exists  $j' \in \{1, \dots, n\}$  such that  $b_{j'} = \sqcup \{b_j \mid j \in J_{d'}\}$  and  $e_{j'} = \sqcup \{e_j \mid j \in J_{d'}\}$ . Clearly any compact element can be represented by a complete listing (as the least upper bound of a finite set of compact elements is compact).

For a complete listing  $[b_1, e_1] \sqcup \dots \sqcup [b_n, e_n]$  of a compact element in  $\llbracket t_1 \rrbracket(\mathcal{I}) \rightarrow \llbracket t_2 \rrbracket(\mathcal{I})$  we now calculate

$$\begin{aligned} & \beta[t_1 \rightarrow t_2]([b_1, e_1] \sqcup \dots \sqcup [b_n, e_n]) = \\ & \lambda p. \sqcup \{ \beta[t_2]([b_1, e_1] \sqcup \dots \sqcup [b_n, e_n])(b) \mid \beta[t_1](b) \sqsubseteq p \wedge b \text{ is compact} \} = \end{aligned}$$

(as the listing is complete and  $\beta[t_1]$  is strict and continuous)

$$\lambda p. \sqcup_j \sqcup \{ \beta[t_2]([b_j, e_j](b)) \mid \beta[t_1](b) \sqsubseteq p \wedge b \text{ is compact} \} =$$

$$\bigsqcup_j \lambda p. \bigsqcup \{ \beta[t_2](e_j) \mid \beta[t_1](b_j) \sqsubseteq p \} = \\ \bigsqcup_j [\beta[t_1](b_j), \beta[t_2](e_j)]$$

By the induction hypothesis all  $\beta[t_1](b_j)$  and  $\beta[t_2](e_j)$  are compact. To show that the above element is compact we must show that  $\{\beta[t_2](e_j) \mid j \in J_{d'}\}$  has an upper bound when  $d' \in [t_1](\mathcal{J})$  and  $J_{d'} = \{j \mid \beta[t_1](b_j) \sqsubseteq d'\}$ . From the definition of  $\text{lpt}(t_1 \rightarrow t_2)$  we know that  $\text{it}(t_2)$  or  $\text{pt}(t_1)$ . If  $\text{it}(t_2)$  then  $[t_2](\mathcal{J})$  is an algebraic lattice so that the set clearly has an upper bound. If  $\text{pt}(t_1)$  then  $J_{d'} = \{j \mid b_j \sqsubseteq d'\}$  so there is  $j' \in \{1, \dots, n\}$  such that  $e_{j'} = \bigsqcup \{e_j \mid j \in J_{d'}\}$  given the assumption about ‘complete listing’. It follows that  $\beta[t_2](e_{j'})$  is an upper bound of the set  $\{\beta[t_2](e_j) \mid j \in J_{d'}\}$ . ■

**Remark 3.4.8.** The function  $\beta[t_1 \rightarrow t_2]$  is intended as a transformation from the domain  $[t_1 \rightarrow t_2](\mathcal{I})$  to the domain  $[t_1 \rightarrow t_2](\mathcal{J})$ . As the reader acquainted with category theory [MacLane, 1971] will know any partially ordered set may be regarded as a simple kind of category. In a similar way a continuous (or at least monotonic) transformation between partially ordered sets may be regarded as a covariant functor. With this in mind we may calculate

$$\beta[t_1 \rightarrow t_2](f) = \lambda p. \bigsqcup \{ (\beta[t_2] \circ f)(d) \mid \beta[t_1](d) \sqsubseteq p \} \\ = \text{Lan}_{\beta[t_1]}(\beta[t_2] \circ f)$$

where we use the formula in [MacLane, 1971, Theorem 4.1, page 236] for the *left Kan extension* of  $\beta[t_2] \circ f$  along  $\beta[t_1]$ .

**Remark 3.4.9.** The first component  $\alpha$  of a pair  $(\alpha, \gamma)$  of adjointed functions is often called a *lower adjoint* and the second component  $\gamma$  an *upper adjoint*. In Fact 3.4.2 we said that any lower adjoint is a representation transformation provided we restrict our attention to adjointed pairs of continuous functions. Assume now that the representation transformer  $\alpha$  specifies lower adjoints  $\alpha_i$  and that  $\alpha_{\times}$  and  $\alpha_{\rightarrow}$  preserve lower adjoints. Then also  $\alpha[t]$  will be a lower adjoint whenever  $t$  is level-preserving. Writing  $\gamma[t]$  for the corresponding upper adjoint we then have

$$\alpha[t_1 \rightarrow t_2](f) = \alpha[t_2] \circ f \circ \gamma[t_1]$$

(Here we have used the fact that an upper adjoint is uniquely determined by its lower adjoint, i.e. if  $(\alpha, \gamma_1)$  and  $(\alpha, \gamma_2)$  are adjointed pairs then  $\gamma_1 = \gamma_2$ .)

### 3.4.2 Weak invertibility

To express that  $\beta[t]$  lives up to the intentions we first need to construct a weak notion of inverse.

**Definition 3.4.10.** A function  $f' : D \times E \rightarrow D$  is a *weak inverse* of a function  $f : D \rightarrow E$  and a relation  $R : E \times E \rightarrow \{\text{true}, \text{false}\}$  if

$$f(d) \sqsubseteq e$$

implies

$$\begin{aligned} d &\sqsubseteq f'(d, e) \\ f(f'(d, e)) &R e \end{aligned}$$

for all  $d \in D$  and  $e \in E$ .

Here the intention is that  $f'$  is the “inverse” of  $f$ , or to be more precise, that  $(f, f')$  behaves as much like an adjointed pair of functions as possible. However,  $D$  is not (necessarily) an algebraic lattice and so we cannot find a “best” description of some  $e \in E$ . Rather we must be content with finding a description  $f'(d, e)$  that is ‘close’ to some  $d \in D$  of interest.

We now propose the following definition of a weak inverse  $\beta'[[t]]$  of  $\beta[[t]]$  and  $\leq[[t]]$ :

$$\begin{aligned} \beta'[[A_i]] &\equiv \lambda(d, e). e \\ \beta'[[t_1 \times t_2]] &\equiv \lambda((d_1, d_2), (e_1, e_2)). (\beta'[[t_1]](d_1, e_1), \beta'[[t_2]](d_2, e_2)) \\ \beta'[[t_1 \rightarrow t_2]] &\equiv \lambda(f, g). \lambda d. \beta'[[t_2]](f(d), g(\beta[[t_1]](d))) \\ \beta'[[\underline{A}_i]] &\equiv \lambda(d, e). d \\ \beta'[[t_1 \underline{\times} t_2]] &\equiv \lambda(d, e). d \\ \beta'[[t_1 \rightarrow t_2]] &\equiv \lambda(d, e). d \end{aligned}$$

The behaviour of  $\beta'[[t]]$  is easy to characterize in a few special cases:

**Fact 3.4.11.** If  $t$  is pure then  $\beta'[[t]](d, e) = e$ .

**Fact 3.4.12.** If  $t$  is impure then  $\beta'[[t]](d, e) = d$ .

In the general case we have:

**Lemma 3.4.13.** *The equations for  $\beta'[[t]]$  define a weak inverse of  $\beta[[t]]$  and  $\leq[[t]]$  whenever  $t$  is level-preserving.*

**Proof.** We prove the result by structural induction on  $t$ . The case  $t = A_i$  is straightforward as  $\beta[[A_i]] = \lambda d. d$  and  $\leq[[A_i]] = =$ . The case  $t = t_1 \times t_2$  follows from the induction hypothesis. The case  $t = \underline{A}_i$  is straightforward as  $\leq[[\underline{A}_i]] = \sqsubseteq$ . Also the cases  $t = t_1 \underline{\times} t_2$  and  $t = t_1 \rightarrow t_2$  are straightforward as we have  $\leq[[t]] = \sqsubseteq$ .

It remains to consider the case  $t = t_1 \rightarrow t_2$ . So assume that  $\beta[[t_1 \rightarrow t_2]](f) \sqsubseteq g$ , i.e.

$$\beta[[t_1]](d) \sqsubseteq e \Rightarrow \beta[[t_2]](f(d)) \sqsubseteq g(e) \quad (*)$$

for all  $d$  and  $e$ . To show  $f \sqsubseteq \beta'[[t_1 \rightarrow t_2]](f, g)$  we consider an argument  $d$  and must show

$$f(d) \sqsubseteq \beta'[[t_2]](f(d), g(\beta[[t_1]](d)))$$

and this follows from the induction hypothesis given the assumption (\*). To show that

$$\beta[[t_1 \rightarrow t_2]](\beta'[[t_1 \rightarrow t_2]](f, g)) \leq [[t_1 \rightarrow t_2]] g$$

we let

$$e \leq [[t_1]] e'$$



and must show

$$\bigsqcup \{ \beta[t_2](\beta'[t_1 \rightarrow t_2](f, g)(d)) \mid \beta[t_1](d) \sqsubseteq e \} \leq [t_2] g(e')$$

i.e.

$$\bigsqcup \{ \beta[t_2](\beta'[t_2](f(d), g(\beta[t_1](d)))) \mid \beta[t_1](d) \sqsubseteq e \} \leq [t_2] g(e')$$

We now consider the “alternatives” in the definition of  $\text{lpt}(t_1 \rightarrow t_2)$  one by one. If  $t_1$  is pure the inequality reduces to

$$\beta[t_2](\beta'[t_2](f(e), g(e))) \leq [t_2] g(e)$$

as  $e = e'$ . The desired result then follows from the induction hypothesis. If  $t_2$  is impure the inequality reduces to

$$\beta[t_1](d) \sqsubseteq e \Rightarrow \beta[t_2](\beta'[t_2](f(d), g(\beta[t_1](d)))) \sqsubseteq g(e')$$

So assume that  $\beta[t_1](d) \sqsubseteq e$ . Using (\*) and the induction hypothesis for  $t_2$  we then get

$$\beta[t_2](\beta'[t_2](f(d), g(\beta[t_1](d)))) \sqsubseteq g(\beta[t_1](d))$$

The result then follows as  $\beta[t_1](d) \sqsubseteq e \sqsubseteq e'$ . ■

The main point of the above lemma is to establish the following corollary showing that another definition of  $\beta[t_1 \rightarrow t_2]$  is possible. However, as is evidenced by [Nielson, 1988] the present route presents fewer technical complications.

**Corollary 3.4.14.**  $\beta[t_1 \rightarrow t_2](f) = \lambda p. \bigvee \{ \beta[t_1](f(d)) \mid \beta[t_1](d) \leq [t_1]p \}$  whenever  $t_1 \rightarrow t_2$  is level-preserving and  $\bigvee$  denotes the least upper bound operator wrt.  $\leq [t_2]$ .

**Proof.** We shall write

$$X(f, p) = \{ \beta[t_2](f(d)) \mid \beta[t_1](d) \leq [t_1] p \}$$

and recall the definition of  $Y(f, p)$  given in the proof of Proposition 3.4.7. As  $X(f, p) \subseteq Y(f, p)$  and  $\bigsqcup Y(f, p)$  exists we get that  $\bigsqcup X(f, p)$  exists and  $\bigsqcup X(f, p) \subseteq \bigsqcup Y(f, p)$ . Next let  $\beta[t_1](d) \sqsubseteq p$  and note that by setting  $d' = \beta'[t_1](d, p)$  we have  $d \sqsubseteq d'$  and  $\beta[t_1](d') \leq [t_1] p$ . Hence

$$\beta[t_2](f(d)) \sqsubseteq \beta[t_2](f(d')) \sqsubseteq \bigsqcup X(f, p)$$

so that  $\bigsqcup Y(f, p) \subseteq \bigsqcup X(f, p)$  and thus  $\bigsqcup Y(f, p) = \bigsqcup X(f, p)$ .

Next we shall show that  $\bigsqcup X(f, p)$  is the least upper bound of  $X(f, p)$  wrt.  $\leq [t_2]$ . For this we consider the “alternatives” in the definition of  $\text{lpt}(t_1 \rightarrow t_2)$  one by one. If  $t_1$  is pure the set  $X(f, p)$  equals the singleton  $\{ \beta[t_2](f(p)) \}$  and clearly  $\beta[t_2](f(p))$  is the least upper bound of this set wrt. the suborder  $\leq [t_2]$ . If  $t_2$  is impure the suborder  $\leq [t_2]$  amounts to  $\sqsubseteq$  and then clearly  $\bigsqcup X(f, p)$  is the least upper bound of  $X(f, p)$  wrt.  $\sqsubseteq$ . ■

### 3.4.3 Optimality

It remains to demonstrate that the transformation  $\beta[t]$  has the required properties (whenever  $t$  is a level-preserving type). We express the correct-

ness using a relation  $\mathcal{R}[\![t]\!]$  as defined in the previous subsection and clearly  $\beta$  and  $\mathcal{R}$  have to “cooperate”:

**Definition 3.4.15.** An admissible relation  $R : D \times E \rightarrow \{\text{true}, \text{false}\}$  *cooperates with* a representation transformation  $f : D \rightarrow E$  and an admissible relation  $R' : E \times E \rightarrow \{\text{true}, \text{false}\}$  if

$$R(d, p) \equiv R'(f(d), p)$$

A correctness correspondence  $\mathcal{R}$  *cooperates with* a representation transformer  $\beta$  if

- $\mathcal{R}_i$  cooperates with  $\beta_i$  and  $\sqsubseteq$ , and
- if  $R_i$  cooperates with  $f_i$  and  $\sqsubseteq$  (for  $i=1,2$ ) then
  - $\mathcal{R}_\times(R_1, R_2)$  cooperates with  $\beta_\times(f_1, f_2)$  and  $\sqsubseteq$ , and
  - $\mathcal{R}_\rightarrow(R_1, R_2)$  cooperates with  $\beta_\rightarrow(f_1, f_2)$  and  $\sqsubseteq$

**Example 3.4.16.** The correctness correspondence  $\text{cor}$  of Example 3.3.4 cooperates with the representation transformer  $\text{b}$  of Example 3.4.5.

**Lemma 3.4.17.** *If  $\mathcal{R}$  cooperates with  $\beta$  then*

$$\mathcal{R}[\![t]\!] \text{ cooperates with } \beta[\![t]\!] \text{ and } \leq[\![t]\!]$$

*for all level-preserving types  $t$ .*

**Proof.** We must prove  $\mathcal{R}[\![t]\!](d, p) \equiv (\beta[\![t]\!](d) \leq[\![t]\!] p)$  by structural induction on a level-preserving type  $t$ .

The case  $t = \mathbb{A}_i$  is straightforward as  $\mathcal{R}[\![t]\!] = \equiv$ ,  $\beta[\![t]\!] = \lambda d. d$ , and  $\leq[\![t]\!] = \equiv$ . The case  $t = t_1 \times t_2$  follows from the induction hypothesis and the componentwise definitions of  $\mathcal{R}[\![t_1 \times t_2]\!]$ ,  $\beta[\![t_1 \times t_2]\!]$ , and  $\leq[\![t_1 \times t_2]\!]$ . The case  $t = \mathbb{A}_i$  is immediate from the assumptions. The cases  $t = t_1 \times t_2$  and  $t = t_1 \rightarrow t_2$  are straightforward given the assumptions, the induction hypothesis, and Lemma 3.3.7.

It remains to consider the case  $t = t_1 \rightarrow t_2$ . We first assume that  $\mathcal{R}[\![t_1 \rightarrow t_2]\!](f, g)$  and by the induction hypothesis this amounts to

$$\beta[\![t_1]\!](d) \leq[\![t_1]\!] e \Rightarrow \beta[\![t_2]\!](f(d)) \leq[\![t_2]\!] g(e) \quad (*)$$

for all  $d$  and  $e$ . To show  $\beta[\![t_1 \rightarrow t_2]\!](f) \leq[\![t_1 \rightarrow t_2]\!] g$  we assume that  $e \leq[\![t_1]\!] e'$  and must show

$$\bigvee \{ \beta[\![t_2]\!](f(d)) \mid \beta[\![t_1]\!](d) \leq[\![t_1]\!] e \} \leq[\![t_2]\!] g(e')$$

where we have used Corollary 3.4.14. For this it suffices to show that

$$\beta[\![t_1]\!](d) \leq[\![t_1]\!] e \Rightarrow \beta[\![t_2]\!](f(d)) \leq[\![t_2]\!] g(e')$$

and this follows from  $(*)$  as  $g \leq[\![t_1 \rightarrow t_2]\!] g$  implies  $g(e) \leq[\![t_2]\!] g(e')$ . Next we assume that  $\beta[\![t_1 \rightarrow t_2]\!](f) \leq[\![t_1 \rightarrow t_2]\!] g$ , i.e. that

$$e \leq[\![t_1]\!] e' \Rightarrow \bigvee \{ \beta[\![t_2]\!](f(d)) \mid \beta[\![t_1]\!](d) \leq[\![t_1]\!] e \} \leq[\![t_2]\!] g(e')$$

holds for all  $e$  and  $e'$ . It follows that

$$\beta[\![t_1]\!](d) \leq[\![t_1]\!] e \wedge e \leq[\![t_1]\!] e' \Rightarrow \beta[\![t_2]\!](f(d)) \leq[\![t_2]\!] g(e')$$

and by choosing  $e'=e$  and using the induction hypothesis we have

$$\mathcal{R}[[t_1]](d, e) \Rightarrow \mathcal{R}[[t_2]](f(d), g(e))$$

for all  $d$  and  $e$ . But this amounts to  $\mathcal{R}[[t_1 \rightarrow t_2]](f, g)$ . ■

### 3.4.4 Summary

Let  $\beta : \mathcal{I} \rightarrow \mathcal{J}$ , or  $\beta : \mathcal{I}^t \rightarrow \mathcal{J}^t$  to be precise, be a representation transformer and let  $\hat{\beta}$  be a correctness correspondence that cooperates with  $\beta$ . An example was given in Example 3.4.16 and we should stress that we do not claim that  $\hat{\cdot}$  is a function as it does not seem possible to define  $\hat{\beta}_{\times}$  and  $\hat{\beta}_{\rightarrow}$  from  $\beta_{\times}$  and  $\beta_{\rightarrow}$  in general.

If  $\mathcal{I}$  is an interpretation for TML[lpt, dt] we may define an interpretation  $\beta(\mathcal{I})$  for TML[lpt, dt], called the *induced analysis*, as follows:

- $(\beta(\mathcal{I}))^t = \mathcal{J}^t$ ,
- for each basic expression or combinator  $\psi$  used with type  $t_\psi$ :  
 $(\beta(\mathcal{I}))_\psi^e = \beta[[t_\psi]](\mathcal{I}_\psi^e)$      \(\backslash\)

We can now be assured that

- $\beta(\mathcal{I})$  is correct, i.e.  $\mathcal{I} \hat{\beta} \beta(\mathcal{I})$ ,
- $\beta(\mathcal{I})$  is optimal, i.e.  $\mathcal{I} \hat{\beta} \mathcal{J} \Rightarrow \beta(\mathcal{I}) \leq \mathcal{J}$ .

In both cases we use that  $\mathcal{I} \hat{\beta} \mathcal{K}$  amounts to  $\beta(\mathcal{I}) \leq \mathcal{K}$  given that  $\hat{\beta}$  cooperates with  $\beta$  and that  $\leq$  is a partial order when we restrict our attention to interpretations of TML[lpt, dt].

## 3.5 Expected forms of analyses

Even though the induced analyses of the previous subsection are optimal they are not necessarily in a form where they are practical or even computable. As an example consider composition  $\square$  which is interpreted as  $\mathbf{S}_{\square}^e = \lambda v_1. \lambda v_2. \lambda w. v_1(v_2(w))$  in the lazy standard semantics. The induced version is

$$\begin{aligned} & \beta[[t' \rightrightarrows t''] \rightarrow (t \rightrightarrows t') \rightarrow (t \rightrightarrows t'')] (\mathbf{S}_{\square}^e) = \\ & \lambda v i_1. \sqcup \{ \beta [(t \rightrightarrows t') \rightarrow (t \rightrightarrows t'')] (\mathbf{S}_{\square}^e (v s_1)) \mid \beta [[t' \rightrightarrows t'']] (v s_1) \sqsubseteq v i_1 \} = \\ & \lambda v i_1. \lambda v i_2. \sqcup \{ \beta [[t \rightrightarrows t'']] (\mathbf{S}_{\square}^e (v s_1) (v s_2)) \mid \beta [[t' \rightrightarrows t'']] (v s_1) \sqsubseteq v i_1 \wedge \\ & \quad \beta [[t \rightrightarrows t'']] (v s_2) \sqsubseteq v i_2 \} = \\ & \lambda v i_1. \lambda v i_2. \sqcup \{ \beta [[t \rightrightarrows t'']] (v s_1 \circ v s_2) \mid \beta [[t' \rightrightarrows t'']] (v s_1) \sqsubseteq v i_1 \\ & \quad \wedge \beta [[t \rightrightarrows t'']] (v s_2) \sqsubseteq v i_2 \} \end{aligned}$$

and this is not as easy to implement as one could have hoped for. In the special case where  $\beta$  specifies lower adjoints as in Remark 3.4.9 we have a slightly nicer induced version (writing  $\alpha$  for  $\beta$ ):

$$\begin{aligned} & \alpha[[t' \rightrightarrows t''] \rightarrow (t \rightrightarrows t') \rightarrow (t \rightrightarrows t'')] (\mathbf{S}_{\square}^e) = \\ & \lambda v i_1. \alpha [[ (t \rightrightarrows t') \rightarrow (t \rightrightarrows t'')] (\mathbf{S}_{\square}^e (\gamma [[t' \rightrightarrows t'']] (v i_1))) = \\ & \lambda v i_1. \lambda v i_2. \alpha [[t \rightrightarrows t'']] (\mathbf{S}_{\square}^e (\gamma [[t' \rightrightarrows t'']] (v i_1)) (\gamma [[t \rightrightarrows t'']] (v i_2))) \end{aligned}$$

If we assume that  $\alpha \rightarrow$  is as in Example 3.4.5, i.e.

$$\alpha \rightarrow (f_1, f_2) = \lambda f. \lambda p. \bigsqcup \{ f_2(f(d)) \mid f_1(d) \sqsubseteq p \}$$

then it follows much as in Remark 3.4.9 that  $\alpha \llbracket t \rightarrow t'' \rrbracket (f) = \alpha \llbracket t'' \rrbracket \circ f \circ \gamma \llbracket t \rrbracket$ ,  $\gamma \llbracket t' \rightarrow t'' \rrbracket (f) = \gamma \llbracket t'' \rrbracket \circ f \circ \alpha \llbracket t' \rrbracket$ , and  $\gamma \llbracket t \rightarrow t' \rrbracket (f) = \gamma \llbracket t' \rrbracket \circ f \circ \alpha \llbracket t \rrbracket$ . It follows that

$$\begin{aligned} \alpha \llbracket (t' \rightarrow t'') \rightarrow (t \rightarrow t') \rightarrow (t \rightarrow t'') \rrbracket (\mathbf{S}_{\square}^e) = \\ \lambda v_{i_1}. \lambda v_{i_2}. \alpha \llbracket t'' \rrbracket \circ (\gamma \llbracket t'' \rrbracket \circ v_{i_1} \circ \alpha \llbracket t' \rrbracket) \circ (\gamma \llbracket t' \rrbracket \circ v_{i_2} \circ \alpha \llbracket t \rrbracket) \circ \gamma \llbracket t \rrbracket = \\ \lambda v_{i_1}. \lambda v_{i_2}. (\alpha \llbracket t'' \rrbracket \circ \gamma \llbracket t'' \rrbracket) \circ v_{i_1} \circ (\alpha \llbracket t' \rrbracket \circ \gamma \llbracket t' \rrbracket) \circ v_{i_2} \circ (\alpha \llbracket t \rrbracket \circ \gamma \llbracket t \rrbracket) \end{aligned}$$

but as  $\alpha \llbracket \cdot \rrbracket \circ \gamma \llbracket \cdot \rrbracket$  in general will differ from the identity also this is not as easy to implement as one could have hoped for.

What we shall do instead is to use functional composition in all analyses. This may not be as precise as possible but will be easier to implement and, as we shall show below, will indeed be correct and thus will make applications easier. A similar treatment can be given for the other combinators and we shall consider a few examples in this subsection. Additional examples may be found in Nielson [1989; 1986b].

**Example 3.5.1.** For a lattice interpretation  $\mathcal{J}$  of TML[lpt,dt] we suggest modelling the composition combinator  $\square$  as functional composition, i.e.

$$\mathcal{J}_{\square}^e = \lambda v_1. \lambda v_2. \lambda w. v_1(v_2(w))$$

and we shall say that this is the *expected form* of  $\square$ . Actually the expected form depends on whether we consider forwards or backwards analyses, i.e. whether  $\rightarrow$  is interpreted as  $\rightarrow$  or as  $\leftarrow$ , as is illustrated by Examples 3.2.11 and 3.2.13 but in this subsection we shall only consider forwards analyses.

To demonstrate the correctness of this expected form let  $\mathcal{I}$  be a domain or lattice interpretation with  $\mathcal{I}_{\square}^e = \lambda v_1. \lambda v_2. \lambda w. v_1(v_2(w))$ . Here  $\mathcal{I}$  may be thought of as the standard semantics or some other lattice interpretation that uses the expected form for  $\square$ . We shall then show

$$\begin{aligned} \beta \llbracket (t' \rightarrow t'') \rightarrow (t \rightarrow t') \rightarrow (t \rightarrow t'') \rrbracket (\mathcal{I}_{\square}^e) \\ \leq \llbracket (t' \rightarrow t'') \rightarrow (t \rightarrow t') \rightarrow (t \rightarrow t'') \rrbracket \\ \mathcal{J}_{\square}^e \end{aligned}$$

where  $\beta : \mathcal{I} \rightarrow \mathcal{J}$  is a representation transformer. In analogy with the assumption that  $\rightarrow$  is interpreted as  $\rightarrow$  in both  $\mathcal{I}$  and  $\mathcal{J}$  it is natural to assume that  $\beta \rightarrow$  is as in Example 3.4.5, i.e.

$$\beta \rightarrow (f_1, f_2) = \lambda f. \lambda p. \bigsqcup \{ f_2(f(d)) \mid f_1(d) \sqsubseteq p \}$$

As  $(t' \rightarrow t'') \rightarrow (t \rightarrow t') \rightarrow (t \rightarrow t'')$  is level-preserving it follows that  $t$ ,  $t'$ , and  $t''$  are impure so that  $\leq \llbracket t \rrbracket \equiv \sqsubseteq$ ,  $\leq \llbracket t' \rrbracket \equiv \sqsubseteq$ , and  $\leq \llbracket t'' \rrbracket \equiv \sqsubseteq$ . The desired result then amounts to assuming that

$$\begin{aligned} \beta \llbracket t' \rrbracket (w_i) \sqsubseteq w_j &\Rightarrow \beta \llbracket t'' \rrbracket (v_{i_1}(w_i)) \sqsubseteq v_{j_1}(w_j) \\ \beta \llbracket t \rrbracket (w_i) \sqsubseteq w_j &\Rightarrow \beta \llbracket t' \rrbracket (v_{i_2}(w_i)) \sqsubseteq v_{j_2}(w_j) \\ \beta \llbracket t \rrbracket (w_i) \sqsubseteq w_j \end{aligned}$$

and showing that

$$\beta[\![t'']\!](vi_1(vi_2(wi))) \sqsubseteq vj_1(vj_2(wj))$$

and this is straightforward.

**Example 3.5.2.** For a lattice interpretation  $\mathcal{J}$  of TML[lpt,dt] we suggest that the *expected form* of Fix amounts to a finite, say 27, number of iterations starting with the “most imprecise” property  $\top$ , i.e.

$$\mathcal{I}_{\text{Fix}}^e = \lambda v. \lambda w. (v(w))^{27}(\top)$$

(Recall that if Fix is used with type  $(t \rightarrow t' \rightarrow t') \rightarrow (t \rightarrow t')$  then  $t'$  will be an impure type so that  $\llbracket t' \rrbracket(\mathcal{J})$  is an algebraic lattice and thus  $\top = \bigsqcup \llbracket t' \rrbracket(\mathcal{J})$  does exist.) This choice of expected form differs from the choice made in the standard semantics of Example 3.2.10 (or the detection of signs analysis of Example 3.2.11) and is motivated by the desire to ensure that an analysis by abstract interpretation terminates. However, it means that we have to give separate proofs for the correctness of this expected form with respect to the standard semantics and for the correctness of continuing to use this expected form.

So let  $\mathcal{I}$  be a domain or lattice interpretation with  $\mathcal{I}_{\text{Fix}}^e = \lambda v. \lambda w. \text{FIX}(v(w))$ . Let  $\beta : \mathcal{I} \rightarrow \mathcal{J}$  be a representation transformer with  $\beta \rightarrow$  as in the previous example. We must then show

$$\begin{aligned} & \beta[\![(t \rightarrow t' \rightarrow t') \rightarrow (t \rightarrow t')]\!](\lambda v. \lambda w. \text{FIX}(v(w))) \\ & \leq [\![(t \rightarrow t' \rightarrow t') \rightarrow (t \rightarrow t')]\!] \\ & (\lambda v. \lambda w. (v(w))^{27}(\top)) \end{aligned}$$

where again  $t$  and  $t'$  will be impure so that  $\leq \llbracket t \rrbracket$  and  $\leq \llbracket t' \rrbracket$  both equal  $\sqsubseteq$ . This amounts to assuming

$$\begin{aligned} & \beta[\![t]\!](wi_1) \sqsubseteq vj_1 \wedge \beta[\![t']\!](wi_2) \sqsubseteq vj_2 \Rightarrow \\ & \beta[\![t']\!](vi(wi_1)(wi_2)) \sqsubseteq vj(vj_1)(vj_2) \\ & \beta[\![t]\!](wi) \sqsubseteq vj \end{aligned}$$

and showing

$$\beta[\![t']\!](\text{FIX}(vi(wi))) \sqsubseteq (vj(vj))^m(\top)$$

for  $m=27$ . We shall prove this by induction on  $m$  and the base case  $m=0$  is immediate. The induction step then follows from the induction hypothesis and the fact that  $(vi(wi))(\text{FIX}(vi(wi)))$  equals  $(\text{FIX}(vi(wi)))$ .

Next let  $\mathcal{K}$  be a lattice interpretation that uses the expected form for Fix and let  $\beta : \mathcal{J} \rightarrow \mathcal{K}$  be a representation transformer with  $\beta \rightarrow$  as above. We must then show

$$\begin{aligned} & \beta[\![(t \rightarrow t' \rightarrow t') \rightarrow (t \rightarrow t')]\!](\lambda v. \lambda w. (v(w))^{27}(\top)) \\ & \leq [\![(t \rightarrow t' \rightarrow t') \rightarrow (t \rightarrow t')]\!] \\ & (\lambda v. \lambda w. (v(w))^{27}(\top)) \end{aligned}$$

This amounts to assuming



$$\begin{aligned}\beta[t](wj_1) &\sqsubseteq wk_1 \wedge \beta[t'](wj_2) \sqsubseteq wk_2 \Rightarrow \\ &\beta[t'](vj(wj_1)(wj_2)) \sqsubseteq vk(wk_1)(wk_2) \\ \beta[t](wj) &\sqsubseteq wk\end{aligned}$$

and showing

$$\beta[t']((vj(wj))^m(\top)) \sqsubseteq (vk(wk))^m(\top)$$

for  $m=27$ . This is once again by induction on  $m$ .

**Example 3.5.3.** In TML[lpt,dt] the meaning of `fix` remains constant, i.e. `fix` is not interpreted by an interpretation but always amounts to the least fixed point `FIX`. Consider now a version of TML where the meaning of `fix` is not constant and thus must be given by an interpretation. When `fix` is used with type  $(t \rightarrow t) \rightarrow t$  this means that we will have to restrict  $(t \rightarrow t) \rightarrow t$  to be level-preserving. It is straightforward to verify that this means that  $t$  must be either pure or impure. When  $t$  is pure it is natural to let an interpretation  $\mathcal{J}$  use the expected form

$$\mathcal{J}_{\text{fix}}^e = \lambda v. \text{FIX}(v)$$

whereas when  $t$  is impure it is natural to let  $\mathcal{J}$  use the expected form

$$\mathcal{J}_{\text{fix}}^e = \lambda v. v^{27}(\top).$$

The correctness of this is shown in [Nielson, 1989].

**Example 3.5.4.** For a lattice interpretation  $\mathcal{J}$  of TML[lpt,dt] we suggest that the *expected form* of `If` amounts to simply combining the effects of the true and else branches, i.e.

$$\mathcal{J}_{\text{If}}^e = \lambda v_1. \lambda v_2. \lambda v_3. \lambda w. (v_2(w)) \sqcup (v_3(w))$$

This is slightly coarser than what we did in Examples 3.2.11 and 3.2.14 in that  $v_1$  is not taken into account but this is in agreement with common practice in data-flow analysis [Aho *et al.*, 1986].

To show the correctness of this let  $\mathcal{I}$  be a domain interpretation along the lines of the lazy standard semantics, i.e.

$$\mathcal{I}_{\text{bool}}^t = A_{\text{bool}}$$

$$\mathcal{I}_{\text{If}}^e = \lambda v_1. \lambda v_2. \lambda v_3. \lambda w. \begin{cases} v_2(w) & \text{if } v_1(w)=\text{true} \\ v_3(w) & \text{if } v_1(w)=\text{false} \\ \perp & \text{if } v_1(w)=\perp \end{cases}$$

(as well as  $\mathcal{I}_{\rightarrow}^t = \rightarrow$  as is the case for all interpretations considered in this subsection). Also let  $\beta : \mathcal{I} \rightarrow \mathcal{J}$  be a representation transformer with  $\beta \rightarrow$  as in the previous example. We must then show

$$\begin{aligned}&\beta[t \rightarrow \text{Bool}] \rightarrow (t \rightarrow t') \rightarrow (t \rightarrow t') \rightarrow (t \rightarrow t') \sqcup (\mathcal{I}_{\text{If}}^e) \\ &\leq [t \rightarrow \text{Bool}] \rightarrow (t \rightarrow t') \rightarrow (t \rightarrow t') \rightarrow (t \rightarrow t') \sqcup \\ &(\mathcal{J}_{\text{If}}^e)\end{aligned}$$

and this follows much as in Example 3.3.4 so we dispense with the details.

Next let  $\mathcal{K}$  be a lattice interpretation that also uses the expected form for **If**. Then we must show

$$\begin{aligned} & \beta[(t \rightarrow \text{Bool}) \rightarrow (t \rightarrow t') \rightarrow (t \rightarrow t') \rightarrow (t \rightarrow t')] (\lambda v_1. \lambda v_2. \lambda v_3. \lambda w. \\ & \quad (v_2(w)) \sqcup (v_3(w))) \\ & \leq [(t \rightarrow \text{Bool}) \rightarrow (t \rightarrow t') \rightarrow (t \rightarrow t') \rightarrow (t \rightarrow t')] \\ & \quad (\lambda v_1. \lambda v_2. \lambda v_3. \lambda w. (v_2(w)) \sqcup (v_3(w))) \end{aligned}$$

and for this we shall need to assume that  $\beta$  specifies lower adjoints as in Remark 3.4.9. Then  $\beta[t']$  is (binary) additive and it is then straightforward to show the desired result.

**Example 3.5.5.** For a lattice interpretation  $\mathcal{J}$  of TML[lpt,dt] the expected forms of the combinators **Tuple**, **Fst**, and **Snd** depend on how  $\times$  is interpreted. When  $\times$  is the tensor product  $\otimes$  one may suggest expected forms based on the definitions given Example 3.2.14 but as we have not covered the tensor product in any detail we shall not look further into this here. We thus concentrate on the case where  $\times$  is interpreted as cartesian product and here we suggest

$$\begin{aligned} \mathcal{J}_{\text{Tuple}}^e &= \lambda v_1. \lambda v_2. \lambda w. (v_1(w), v_2(w)) \\ \mathcal{J}_{\text{Fst}}^e &= \lambda v. \lambda w. d_1 \text{ where } (d_1, d_2) = v(w) \\ \mathcal{J}_{\text{Snd}}^e &= \lambda v. \lambda w. d_2 \text{ where } (d_1, d_2) = v(w) \end{aligned}$$

For correctness we shall assume that  $\mathcal{I}$  is a domain or lattice interpretation that uses “analogous definitions”, either because it is the standard semantics, or because it is some lattice interpretation that also uses the expected forms. Furthermore we shall assume that  $\beta : \mathcal{I} \rightarrow \mathcal{J}$  is a representation transformer with  $\beta \rightarrow$  as in the previous examples and with  $\beta_\times$  as in Example 3.4.5, i.e.

$$\beta_\times(f_1, f_2) = \lambda(d_1, d_2). (f_1(d_1), f_2(d_2))$$

The correctness is then expressed and proved using the pattern of the previous examples and we omit the details.

**Example 3.5.6.** For a lattice interpretation  $\mathcal{J}$  of TML[lpt,dt] we shall suggest the following expected forms for **Curry** and **Apply**:

$$\begin{aligned} \mathcal{J}_{\text{Curry}}^e &= \lambda v. \lambda w_1. \lambda w_2. v(w_1, w_2) \\ \mathcal{J}_{\text{Apply}}^e &= \lambda v_1. \lambda v_2. \lambda w. v_1(w)(v_2(w)) \end{aligned}$$

Here we assume that  $\rightarrow$  is interpreted as  $\rightarrow$  and that  $\times$  is interpreted as cartesian product  $\times$ . To show correctness we consider a domain or lattice interpretation  $\mathcal{I}$  that uses “analogous definitions”. Concerning the representation transformer  $\beta : \mathcal{I} \rightarrow \mathcal{J}$  we shall assume that  $\beta \rightarrow$  and  $\beta_\times$  are as in the previous example. The correctness is then expressed and proved using the pattern of the previous examples and we omit the details.

These examples have illustrated that the practical application of a framework for abstract interpretation may be facilitated by studying certain expected forms for the combinators. In this way one obtains correct and implementable analyses that can be arranged always to terminate although at the expense of obtaining more imprecise results than those guaranteed by the induced analyses.

### 3.6 Extensions and limitations

In this section we have aimed at demonstrating the main approach, the main definitions, the main theorems, and the main proof techniques employed in a framework for abstract interpretation. To do so we have used a rather small metalanguage based on the typed  $\lambda$ -calculus and in this subsection we conclude by discussing possible extensions and current limitations.

The discussion will centre around [Nielson, 1989]. The metalanguage  $\text{TML}_m$  considered there has sum types and recursive types in addition to the base types, product types, and function types. However, the well-formedness conditions imposed on the two-level types in [Nielson, 1989] are somewhat more demanding than those imposed here. Let  $\text{mc}(t)$  denote the condition that every underlined type constructor in  $t$  has arguments that are all-underlined and that any underlined construct occurs in a subtype of the form  $t' \rightarrow t''$ . Then the fragment of  $\text{TML}_m$  [Nielson, 1989] that only has base types, product types, and function types corresponds to  $\text{TML}[\text{lpt} \wedge \text{mc}, \text{dt} \wedge \text{mc}]$ <sup>7</sup> rather than the  $\text{TML}[\text{lpt}, \text{dt}]$  studied here. As an example this means that a type like  $(\underline{A}_{\text{int}} \rightarrow \underline{A}_{\text{int}}) \times \underline{A}_{\text{bool}}$  is well-formed in  $\text{TML}[\text{lpt}, \text{dt}]$  but not in  $\text{TML}[\text{lpt} \wedge \text{mc}, \text{dt} \wedge \text{mc}]$ .

Concerning sum types one can perform a development close to that performed for product types. In a sense an analogue to the relational method is obtained by modelling  $\perp$  as cartesian product whereas an analogue of the independent attribute method is obtained by interpreting  $\perp$  as a kind of sum (adapted to produce an algebraic lattice). For recursive types there are various ways of solving the recursive type equations and one of these amounts to truncating a recursive structure at a fixed depth. Turning to expressions an analysis like detection of signs is formulated and proved correct for the whole metalanguage and also strictness analysis can be handled [Nielson, 1988]. It is also shown that induced analyses exist in general and this is used to give a characterization of the role of the *collecting semantics* (*accumulating standard semantics* in the terminology of the Glossary). For a second-order and backwards analysis like live variables analysis a formulation much like the one given in Example 3.2.13 is proved correct. As we already said in section 3.1 this may be extended with several *versions* of underlined type constructors without a profound change in the theory.

---

<sup>7</sup>One may verify that  $\text{dt} \wedge \text{mc}$  is equivalent to  $\text{mc}$ .

Even for  $\text{TML}[\text{lpt}\wedge\text{mc}, \text{dt}\wedge\text{mc}]$  the development in [Nielson, 1989] goes a bit further than overviewed here. The presence of the constraint **mc** makes it feasible to study interpretations, so-called *frontier interpretations*, where the interpretation of underlined types is *not* specified in a structural way. (In a sense one considers a version of the metalanguage without  $\underline{x}$  and  $\underline{\rightarrow}$  but with a greatly expanded index set  $\mathbf{I}'$  over which the index  $i$  in  $\underline{A}_i$  ranges.) This makes it feasible to give a componentwise definition of the composition  $\beta' \circ \beta$  of frontier representation transformers. This is of particular interest when  $\beta'$  only specifies representation transformations that are lower adjoints. Writing  $\beta' = \alpha$  one can then show that  $(\alpha \circ \beta)[t] = \alpha[t] \circ \beta[t]$  for level-preserving types  $t$  and this may be regarded as the basis for developing abstract interpretations in a stepwise manner. Also the transformation from  $\beta$  to  $\hat{\beta}$  becomes functional and one can show that  $\alpha \hat{\circ} \beta = \hat{\beta} \circ \hat{\alpha}$ .

From a practical point of view a main limitation is that we have not incorporated *stickiness*, i.e. we transform a property through a program but we do not record the properties that reach a given program point. Such a development would be very desirable when one wants to exploit the results of an analysis to enable *program transformations* that are not valid, i.e. meaning preserving, in general. This is illustrated in [Nielson, 1985b] that uses the results of analyses as specified in [Nielson, 1982]. It applies equally well to the flow analysis techniques used in practical compilers.

However, it is not a minor task to incorporate this into the present framework. One reason is that the way it is done depends heavily on details of the *evaluation order* used in an implementation and these details are not specified by the standard semantics.

## 4 Other analyses, language properties, and language types

In the first sections we described the roots of abstract interpretation and gave a motivated development of the Cousot approach. We then showed how their methods can be systematically extended to languages defined by denotational semantics, using logical relations to lift approximations from base domains to product and function space domains. This extension allows analysis of a wide range of programming languages, by abstractly interpreting the domains and operations appearing in the denotational semantics that defines them.

In section 3 the denotational approach was generalized even more: we rigorously developed an abstract interpretation framework based on reinterpreting some of the primitive operations appearing in *the metalanguage* used to write denotational language definitions. This yields a framework that is completely independent of any particular programming language.



We now describe some other analysis methods, language properties, and language types. As to methods, we will see that it is sometimes necessary to *instrument* a semantics to make it better suited for analysis, i.e. to modify it so it better models operational aspects relevant to program optimization. (A related approach, not yet as fully developed, is to derive analyses from operational semantics rather than denotational ones.)

Abstract interpretation has its roots in applications to optimizing compilers for imperative languages, so it is not surprising that the early papers by Cousot on semantically based methods were about such programs. A later wave of activity concerned efficient implementation of high-level functional languages. Recent years have witnessed a rapid growth of research in abstractly interpreting logic programming languages, Prolog in particular. Analysis of both functional and logic programming languages will be discussed briefly.

In contrast to the previous section we only give an overview of basic ideas, motivations, and a few examples, together with some references to the relevant literature (large—a bibliography from 1986 may be found in [Nielson, 1986a]).

## 4.1 Approaches to abstract interpretation

We now briefly assess what we did in earlier sections, and give some alternative approaches to program analysis by abstract interpretation. Each has some advantages and disadvantages.

### 4.1.1 The Cousot approach

This was the first truly semantics-based framework for abstract interpretation, and had many ideas that influenced development of the field, for example abstraction and concretization functions, natural mathematical conditions on them, and the collecting (or static) semantics. The collecting semantics is “sticky”, meaning that it works by binding information describing the program’s stores to program points. This provided a natural link to the earlier and more informal flow analysis methods used in optimizing compilers, based on constructing and then solving a set of “data-flow equations” from the program.

The abstraction and concretization functions are a pair of functions  $\alpha : C \rightarrow \text{Abs}$  and  $\gamma : \text{Abs} \rightarrow C$  between concrete values  $C$  and abstract values  $\text{Abs}$ . Both are required to be complete lattices, and  $\alpha, \gamma$  must satisfy some fairly stringent conditions ( $\gamma$  must be a Galois insertion from  $\text{Abs}$  into  $C$ ).

An important new concept was that of a *program interpretation*, abstracting the program’s stores. A partial order on interpretations was defined, making it possible to prove rigorously that one interpretation is a correct abstraction of another. This allows program analysis methods to be proven “safe”, i.e. to be correct approximations to actual program behaviour.



A limitation is that the Cousot approach only applies to flow chart programs, and has been difficult to extend to, for example, programs with procedures (examples include [Jones and Muchnick, 1982] and [Sharir and Pnueli, 1981]). Another limitation is in its data: as originally formulated, only stores were abstracted, and no systematic way to extend the framework to more general data types was given.

#### 4.1.2 Logical relations

This solved the problem of extending the Cousots' methods to more general data, using logical relations as defined in [Reynolds, 1974], [Plotkin, 1980]. (In the discussion above on local conditions for safety, the logical relation was  $\leq_\beta$  on values.) Safe approximation of composite domains is defined by induction on the form of the domain definitions, leading to a natural sufficient condition for safety of an abstract interpretation that generalizes the Cousots' formulation. In this approach, concretization is not mentioned at all, nor does it seem to be necessary.

Example works using this approach are [Mycroft and Jones, 1986], [Jones and Mycroft, 1986], and [Nielson, 1984].

#### 4.1.3 A method based on a metalanguage

The previous method applies only to one language definition at a time. Yet more generality can be obtained by abstractly interpreting the metalanguage used to write the denotational semantics (typically the lambda calculus). This is done in [Nielson, 1984] and subsequent papers, and is summarized in section 3. A two-level lambda calculus is used to separate those parts of a denotational semantics that are to be approximated from those to remain uninterpreted.

The approach seems to be inherently "non-sticky", as it concerns approximating intermediate values and lacks a means for talking about program points.

#### 4.1.4 Operationally based methods

A major purpose of abstract interpretation is its application to efficient program implementation, e.g. in highly optimizing compilers. For application purposes, it thus seems more relevant to use analyses based on a semantics of the language being analysed, rather than on the metalanguage in which the semantics is written.

But there is a fly in the ointment: many implementation-dependent properties relevant to program optimization are simply *not present* in a standard denotational semantics. (This is not at all surprising, if we recall that the original goal of denotational semantics [Stoy, 1977] was to assign the right input-output behaviour to the programs *without* giving irrelevant implementation details.) Examples include:

- the dependence analyses mentioned in the first section (for neededness

analysis, or partial evaluation)

- sequential information about values, e.g. that a variable grows monotonically
- order of parameter evaluation
- time or space usage
- available expressions.

Interesting properties that can be extracted from a denotational semantics include strictness analysis and a (rather weak form of) termination [Burn *et al.*, 1986], [Abramsky, 1990]. In the approach of [Abramsky, 1990] this is done by focusing on logical relations and then developing “best interpretations” with respect to these: the notion of safety leads to a strictness analysis, whereas the dual notion of liveness leads to a termination analysis. This is all related to adjunctions between categories and the use of the formula for Kan extensions. However, while a compositional strictness analysis is indeed useful, a compositional termination analysis is not, because it has to “give up” for recursion; amending this would entail finding a well-founded order with respect to which the recursive calls do decrease and this is beyond the development of [Abramsky, 1990].

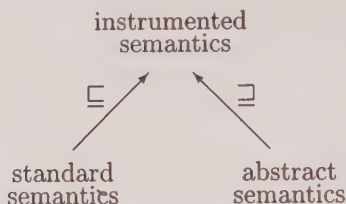
*Operational semantics* It would seem obvious to try to extract these properties from an *operational semantics* [Plotkin, 1981], [Kahn, 1987]. However, this is easier said than done, for several reasons. One is that there is no clearly agreed standard for what is allowed to appear in an operational semantics, other than that it is usually given by a set of conditional logical inference rules (for instance, single-valuedness of an expression evaluation function must be proven explicitly). Another is that operational semantics do not by their nature give all desirable information, e.g. the resource usage or dependency information mentioned above.

It must be said, though, that operational semantics has a large potential as a basis for abstract interpretation, since it more faithfully models actual computational processes, as evidenced by the many unresolved problems concerning full abstraction in denotational semantics. Further, operational semantics is typically restricted to first-order values and so avoids sometimes painful questions involving Scott domains. For instance, there are still several open questions about how to approximate power domains for abstract interpretation of nondeterministic programs.

#### 4.1.5 Instrumented semantics

Much early work in static program analysis was based on approximating informal models of program execution—and was complex and sometimes wrong. On the other hand, the validity of certain program optimizations and compilation techniques may depend strongly on execution models, e.g. some of the properties just mentioned.

A denotationally based method to obtain information about program execution is to *instrument* the standard semantics, extending it to include additional detail, perhaps operational. The approach can be described by the following diagram, where the left arrow follows since the instrumented semantics extends the standard one, and the right one comes from the requirement of safe approximation. On the other hand all three are obtained by various interpretations of the core semantics, so analysis is still done by abstractly executing source programs.



The collecting semantics illustrates one way to instrument, by collecting the state sets at program points. This is practically significant since many interesting program properties are functions of the sets of states that occur at the program's control points.

More general instrumentation could record a trace or history of the entire computation, properties of the stores or environments, forward or backward value dependencies, sequences of references to variables, and much more. This could be used to collect forward dependence information, monotone value growth, or perform step counting. The sketched approach puts the program flow analyses used in practice on firmer semantical foundations.

The absence of an arrow between the standard and the abstract semantics is disturbing at first, since correctness is no longer simply a matter of relating abstract values to the concrete ones occurring in computations. However, this is inevitable, once the need to incorporate some level of operational detail has been admitted. One must be sure that the extension of the standard semantics properly models the implementation techniques on which optimization and compilation can be based; and this cannot be justified on semantical grounds alone.

On the other hand, an approximate semantics can be proven correct with respect to the instrumented version by exactly the methods of the previous sections.

## 4.2 Examples of instrumented semantics

The possible range of instrumented semantics is enormous, and many variants have already been invented for various optimization purposes. Here we give just a sampling.

#### 4.2.1 Program run times

A program time analysis can be done by first extending the standard semantics to record running times, and then to approximate the resulting instrumented semantics. Here we use the denotational framework of section 2.7. The earlier semantics is extended by accumulating, together with the store, the time since execution began.

*The time interpretation* This is  $\mathbf{I}_{time} = (\text{Val}, \text{Sto}; \text{assign}, \text{seq}, \text{cond}, \text{while})$ , defined by

##### Domains

$\text{Val} = \text{Number}$  (the flat cpo)  
 $\text{Sto} = (\text{Var} \rightarrow \text{Val}) \times \text{Number}$

##### Function definitions

$\text{assign} = \lambda(x, m_e) . \lambda(s, t) . (s[x \mapsto m_e s], t+1)$   
 $\text{seq} = \lambda(m_{1c}, m_{2c}) . m_{2c} \circ m_{1c}$   
 $\text{cond} = \lambda(m_e, m_{1c}, m_{2c}) . \lambda(s, t) .$   
 $\quad m_e s \neq 0 \rightarrow m_{1c}(s, t+1), m_{2c}(s, t+1)$   
 $\text{while} = \lambda(m_e, m_c) . \text{fix } \lambda\phi . \lambda(s, t) .$   
 $\quad m_e s \neq 0 \rightarrow \phi(m_c(s, t+1)), (s, t+1)$

This was used as the basis for the approximate time analyses reported in [Rosendahl, 1989].

#### 4.2.2 Execution traces

A closely related idea is to instrument a semantics by including full computation histories. This gives in a sense all the raw material that can be used to extract “history-dependent” information, and thus a basis for a very wide range of program analyses. This approach was used in P. Cousot’s thesis work, and has since been seen in [Donzeau-Gouge, 1981] and [Nielson, 1982].

In a functional setting, Sestoft has traced sequences of variable definitions and uses (i.e. bindings and references) in order to see which variables can be “globalized”, i.e. allowed to reside in global memory instead of the computation stack [Sestoft, 1988]. A similar idea is used in [Bloss and Hudak, 1986] for efficient implementation of lazy functional languages; they trace references to functions’ formal parameters to see which ones can be computed using call by value, i.e. prior to function entry.

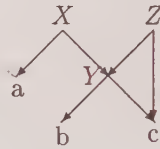
#### 4.2.3 Store properties

The sequencing information just mentioned is quite clearly not present in a standard semantics. Another information category very useful for optimization has to do with store properties. A standard semantics for a language with structured values often treats them simply as trees, i.e. terms;



but in reality memory sharing is used, so new terms are constructed using pointers to old ones rather than a very expensive recopying. For efficient implementation, compile-time analyses must take sharing into account, for example to minimize costs of memory allocation and garbage collection. (Careful proofs of equivalence between the two formulations for a term rewriting language can be seen in [Barendregt, 1989].)

In [Jones and Muchnick, 1981] a simple imperative language with Lisp-like primitives is discussed. The semantics is described operationally, using finite graphs for the store. The following diagram describes a store in which  $X = a :: (b :: c)$ ,  $Y = b :: c$ , and  $Z = (b :: c) :: c$ , where  $Y$  is shared by  $X$  and  $Z$ .



This instrumented semantics is modelled in [Jones and Muchnick, 1981] by approximating such stores by “k-limited graphs”, where  $k$  is a distance parameter. The idea is that the graph structure is modelled exactly for nodes of distance  $k$  or less from a variable. All graph nodes farther than  $k$  from a variable name are modelled by nodes of the forms “?c”; “?s”; or “?”, indicating that the omitted portion of the graph is (respectively) possibly cyclic; acyclic but with possibly shared nodes; or acyclic and without sharing.

Stransky has done further work in this direction [Stransky, 1992].

### 4.3 Analysis of functional languages

Motivations for analysing functional languages are partly to counter the time costs involved in implementing powerful programming features such as higher-order functions, pattern matching, and lazy evaluation; and partly to reduce their sometimes large and not easily predictable space requirements.

The problem of strictness analysis briefly mentioned in section 1.1 has received much attention, key papers being [Burn *et al.*, 1986], [Wadler and Hughes, 1987], and [Hughes and Launchbury, 1992]. Polymorphism, which allows a function’s type to be used in several different instantiations, creates new problems in abstract interpretation. Important papers include [Abramsky, 1986] and [Hughes and Launchbury, 1992].

*Approximating functions by functions on abstract values* A natural and common approach is to let an abstraction of a function be a function on



abstract values. One example is section 3's general framework using logical relations and based on the lambda calculus as a metalanguage.

The first higher-order strictness analysis was the elegant method of [Burn *et al.*, 1986]. In this work the domains of a function being analysed for strictness are modelled by abstract domains of exactly the same structure, but with  $\{\perp, \top\}$  in place of the basis domains. Strictness information is in essence obtained by computing with these abstracted higher-order functions. However, a fixpoint iteration is needed, since the abstractions sacrifice the program's determinacy (cf. the end of section 1.1).

While suitable for many problems concerning functional and other languages, abstracting functions by functions is not always enough for the program analyses used in practice. An example where this simply does not give enough information is *constant propagation* in a functional language. As before, the goal is to determine whether or not one of the arguments of a program function is always called with the same value and, if so, to find that value at analysis time.

For this it is not enough to know that if a function  $f$  is called with argument  $x$ , it will return value  $x + 1$ . It is also essential to know *which values  $f$  can be called with* during program execution, since a compiler can exploit knowledge about constant arguments to generate better target code.

For another example, the higher-order strictness analysis in [Burn *et al.*, 1986] has turned out to be unacceptably slow in practice, even for rather small programs. The reason is that the abstract interpretation involves computing fixpoints of higher-order functions. Abstract domains for the function being analysed have the same structure as the originals, but with  $\{\perp, \top\}$  for all basis domains. This implies that analysis of even a small function such as "fold right" can lead to a combinatorial explosion in the size of the abstract domains involved, requiring subtle techniques to be able to compute the desired strictness information and avoid having to traverse the entire abstract value space.

#### 4.3.1 First-order minimal function graphs

The *minimal function graph* of a program function was defined in [Jones and Mycroft, 1986] to be the smallest set of pairs (argument, function value) sufficient to carry out program execution on given input data. For example, consider the function defined by the following program:

$$\begin{aligned} f(X) = & \text{ if } X = 1 \text{ then } 1 \\ & \text{ else if } X \text{ even then } f(X/2) \\ & \text{ else } f(3 * X + 1) \end{aligned}$$

Its minimal function graph for program input  $X = 3$  is

$$\{(3, 1), (10, 1), (5, 1), (16, 1), (8, 1), (4, 1), (2, 1), (1, 1)\}$$

The minimal function graph semantics maps a programmer-defined function to something more detailed than the argument-to-result function traditionally used in a standard semantics, and is a form of instrumented semantics. In [Jones and Mycroft, 1986] it is shown how the “constant propagation” analysis may be done by approximating this semantics, and the idea of proving correctness by semihomomorphic mappings between various interpretations of a denotational semantics is explained.

Earlier methods to approximate programs containing function calls were described in [Cousot and Cousot, 1977c], [Sharir and Pnueli, 1981], and [Jones and Muchnick, 1982].

### 4.3.2 Higher-order functions

Higher-order functions as well as first-order ones may be approximated using logical relations. Examples include the analyses of section 3.2.3 and [Burn *et al.*, 1986], but such methods cannot closely describe the way functions are used during execution.

For example, if the value of *exp* in an application *exp(exp')* is modelled by a function from abstract values to abstract values, this does not contain enough information to see just *which programmer-defined functions* may be called at run time; and such information may be essential for efficient compilation.

This leads to more operational approaches to abstractly interpreting programs containing higher-order functions. An early step in this direction was the rather complex [Jones, 1981], and more recent and applications-motivated papers include [Sestoft, 1988; Shivers, 1988].

*Closure analysis* This method from [Sestoft, 1988] can be described as an operationally oriented semantics-based method. Programs are assumed given as systems of equations in the now popular named combinator style, with all functions curried, and function definitions of the form

$$f X_1 \dots X_n = \text{expression containing } X_1 \dots X_n \text{ and function names}$$

In this language, a value which is a function is obtained by an incomplete function application. Operationally such a value is a so-called *closure* of form  $\langle f v_1 \dots v_i \rangle$  where  $v_1 \dots v_i$  are the values of *f*'s first *i* arguments.

The paper [Sestoft, 1988] contains algorithms for a closure analysis, yielding for example information that in a particular application *exp(exp')*, the operator *exp* can evaluate to an *f*-closure with one evaluated argument, or to a *g*-closure with two evaluated arguments.

Instead of approximating a function by a function, each programmer-defined function *f* is described by a *global function description table* con-

taining descriptions of all arguments with which it can be called (just as in the minimal function graphs discussed earlier).

An approximation to a value which is a function is thus represented by an approximate closure of the form  $\langle fa_1 \dots a_i \rangle$  where the  $a_i$  approximate the  $v_i$ .

But then how are the  $v_i$  themselves approximated? (There seems to be a risk of infinite regression.) Supposing  $v_i$  can be a closure  $\langle gw_1 \dots w_j \rangle$ , we can simply approximate it by the pair  $\langle g, j \rangle$ . The reason this works is that, when needed, a more precise description of  $g$ 's arguments can be obtained from  $g$ 's entry in the global function description table.

Analysis starts with a single global function description table entry describing the program's initial call, and abstract interpretation continues until this table stabilizes, i.e. reaches its fixpoint. Termination is guaranteed because there are only finitely many possible closure descriptions.

Closure analysis describes functions globally rather than locally, and so appears to be less precise in principle than approximating functional values by mathematical functions. This is substantiated by complexity results that show the analyses of [Burn *et al.*, 1986] to have a worst-case lower bound of exponential time, whereas closure analysis works in time bounded by a low-degree polynomial.

The techniques developed in [Sestoft, 1988; Shivers, 1988] have shown themselves useful for a variety of practical flow analysis problems involving higher-order functions, for example in efficient implementation of lazy evaluation [Sestoft and Argo, 1989] and partial evaluation [Bondorf, 1990; Bondorf, 1991]. Similar ideas are used in [Jones and Rosendahl, 1994] to provide an operationally oriented denotational minimal function graph semantics for higher-order programs. One application is to prove the safety of Sestoft's algorithms.

### 4.3.3 Backwards analysis and contexts

As observed in section 2.5.3, backwards analysis of an imperative program amounts to finding the weakest precondition on stores sufficient to guarantee that a certain postcondition will hold after command execution. For functional programs, an analogous concept to postcondition is that of the *context* of a value, which describes the way the value will be used in the remainder of the computation.

Clearly the usage of the result of a function will affect the usage of the arguments to the function. For an extreme example, if the result of function call  $f(e_1, \dots, e_n)$  is not needed, then the arguments  $e_1, \dots, e_n$  will not be needed either.

The example is not absurd; consider the following abstract program, using pattern matching and a list notation where  $\text{nil} = []$  is the empty list,  $:$  is the concatenation operator,  $[a_1, \dots, a_n]$  abbreviates  $a_1 : \dots : a_n : \text{nil}$ .

$$\begin{aligned}
 \text{length}(\square) &= 0 \\
 \text{length}(Z:Zs) &= 1 + \text{length}(Zs) \\
 f(X) &= \text{if } \text{test}(X) \text{ then } \square \text{ else } g(X) : f(X-1)
 \end{aligned}$$

When evaluating a call  $\text{length}(f(\text{exp}))$ , the values of  $g(\dots)$  are clearly irrelevant to the length of  $f(\text{exp})$ , and  $g$  need not be called at all. (This can be used to optimize code in a lazy language.) For another example, if  $f(n, x) = x^n$  and a call  $f(e_1, e_2)$  appears in a context where its value is an even number, one can conclude that  $e_1$  is positive and  $e_2$  is even.

Context information thus propagates backwards through the program: from the context of an enclosing expression to the contexts of its subexpressions, and from the context of a called function to the contexts of its parameters in a call.

#### *Some uses of backwards functional analyses*

*Strictness analysis* identifies arguments in a lazy or call by name language for which the more efficient call by value evaluation may be used without changing semantics. Both forwards and backwards algorithms exist, but backwards methods seem to be faster. Early work on backwards methods includes [Hughes, 1986] and [Hughes, 1987], later simplified for the case of domain projections in [Wadler and Hughes, 1987] and [Hughes and Launchbury, 1992].

*Storage reclamation.* Methods are developed in [Jensen and Mogensen, 1990] to recognize when a memory cell has been used for the last time, so it may safely be freed for later use. An application is substantially to reduce the number of garbage collections.

*Partial evaluation* automatically transforms general programs into versions specialized to partially known inputs. A specialized program is usually faster than the source it was derived from, but often contains redundant data structures or unused values (specializations of general-purpose data in the source). Backwards analysis is used for “arity raising”, which improves programs by removing unnecessary data and computation [Romanenko, 1990].

*Information flow* While the analogy to the earlier backwards analyses is clear, the technical details are different and rather more complicated. A bottleneck is that, while functions may have many arguments, they produce only one result. Thus one cannot simply invert the “next” relation to describe program running in the reverse direction.

Given a function definition

$$f X_1 \dots X_n = \text{expression containing } X_1 \dots X_n \text{ and function names}$$

one can associate a context transformer  $f^i : \text{Context} \rightarrow \text{Context}$  with each argument  $X_i$ . The idea is that if  $f$  is called with a result context  $C$ , then  $f^i(C)$  will be the context for  $f$ ’s  $i$ th argument.



In effect this is an independent attribute formulation of  $f$ 's input-output relation, necessarily losing intervariable interactions. Unfortunately it means that backwards analyses cannot in principle exactly describe the program's computations, as is the case with forwards analyses.

*What is a context, semantically?* The intuition “the rest of the computation” can be expressed by the *current continuation*, since a continuation is a function taking the current expression value into the program's computational future. This approach was taken in [Hughes, 1987], but is technically rather complex since it entails that a context is an abstraction of a set of continuations—tricky to handle since continuations are higher-order functions.

In the later [Wadler and Hughes, 1987] and [Hughes and Launchbury, 1992], the concept of context is restricted to properties given by *domain projections*, typically specifying which parts of a value might later be used. A language for finite descriptions of projections and their manipulation was developed, flow equations were derived from the program to be analysed, and their least fixpoint solution gives the desired information.

Some example contexts that have shown themselves useful for the efficient implementation of lazy functional programs include:

- ABSENT: the value is not needed for further computation
- ID: the entire value may be needed
- HEAD: the value is a pair, and its first component may be needed (but the second will not be)
- SPINE: the value is a list, and all its top-level cells may be needed (the *length* function demands a SPINE context of its argument).

## 4.4 Complex abstract values

Finding finite approximate descriptions of infinite sets of values is an essential task in abstract interpretation. We mentioned in Section 2.5 that abstracting stores or environments amounts to finding finite descriptions of relations among the various program variables. This was straightforward in the even-odd example given earlier, e.g. “odd” represented  $\{1,3,5,\dots\}$ , etc., and operations on numbers were easily modelled on this finite domain of abstract values. Analysis problems requiring more sophisticated methods include

- functions as values (especially higher-order functions)
- mutual relationships among variable values
- describing structured data, e.g. nested lists and trees.

### 4.4.1 Functions as values

Some approaches were described above (approximation by functions on abstract values, and closure analysis), and several more have been studied.



#### 4.4.2 Relations on $n$ -tuples of numbers

In this special case there is a well-developed theory: linear algebra, in particular systems of linear inequalities. Cousot and Halbwachs, [1978] describes a way to discover linear relationships among the variables in a Pascal-like program. Such relations may be systematically discovered and exploited for, for example, efficient compilation of array operations. Related work, involving the inference of systems of modulus equations, has been applied to pipelining and other techniques for utilizing parallelism [Granger, 1991; Mercouroff, 1991]. This work has been further developed into a system for automatic analysis of Pascal programs.

#### 4.4.3 Grammars

The analysis of programs manipulating structured data, e.g. lists as in Lisp, ML, etc., requires methods to approximate the infinite sets of values that variables may take on during program runs. There is also a well-developed theory and practice for approximating such infinite sets, involving *regular grammars* or their equivalent, *regular expressions*.

*An example grammar construction* Consider the following abstract program, using the list notation of section 4.3.3.

$f(N)$	$=$	$\text{first}(N, \text{sequence}(\text{nil}))$
$\text{first}(\text{nil}, Xs)$	$=$	$\text{nil}$
$\text{first}(M : Ms, X : Xs)$	$=$	$Ms : \text{first}(Ms, Xs)$
$\text{sequence}(Y)$	$=$	$Y : \text{sequence}(1 : Y)$

We assume an initial call of form  $f(N)$  where the input variable  $N$  ranges over all lists of 1's, and further that the language is lazy. Conceptually, call “ $\text{sequence}(\text{nil})$ ” generates the infinite list  $[], [1], [1,1], [1,1,1], \dots$ . The possible results of the program are all of its finite prefixes:

Output =  $\{[], [[1], [1,1], [1,1,1], [1,1,1,1], [1,1,1,1,1], \dots\}$

The method of [Jones, 1987] constructs from this program a tree grammar  $G$  containing (after some simplification) the following productions. They describe the terms which are the program's possible output values:

$N$	$::=$	$\text{nil} \mid 1 : N$	Program input
$f_{\text{result}}$	$::=$	$\text{first}_{\text{result}}$	
$\text{first}_{\text{result}}$	$::=$	$\text{nil} \mid Ms : \text{first}_{\text{result}}$	
$M$	$::=$	$1$	
$Ms$	$::=$	$N$	
$X$	$::=$	$Y$	
$Xs$	$::=$	$\text{sequence}_{\text{result}}$	
$Y$	$::=$	$\text{nil} \mid 1 : Y$	
$\text{sequence}_{\text{result}}$	$::=$	$Y : \text{sequence}_{\text{result}}$	

With  $f_{result}$  as initial nonterminal,  $G$  generates all possible lists, each of whose elements is a list of 1's. More generally, by this approach, an abstract value in  $Abs$  is a tree grammar, and the concretization function maps the tree grammar and an identified nonterminal symbol  $A_i$  into the set of terms that  $A$  generates.

*Safety* The natural definition of *safe program approximation* is that the grammar generates all possible run-time values computed by the program (usually a proper superset). By the grammar above, nonterminal  $f_{result}$  clearly generates all terms in  $Output$  — and so is a safe approximation to the actual program behaviour.

It is not a perfect description, since  $f_{result}$  generates all possible lists of lists of 1's, regardless of order.

*Nonexistence of an abstraction function  $\alpha$*  For this analysis an abstract value is a large object: a grammar  $G$ . The concretization function  $\gamma$ , maps  $G$ 's nonterminals into term sets which are supersets of the value sets that variables range over in actual computations. It is natural to ask: what is the corresponding abstraction function  $\alpha$ ?

In this case, there is no unique natural  $\alpha$ , for a mathematical reason. The point is that regular tree grammars as illustrated above generate only *regular sets of terms*, a class of sets with well-known properties. On the other hand, the program above, and many more, generate nonregular sets of values ( $Output$  is easily proven nonregular). It is well known that for any nonregular set  $S$  of terms, there is no “best” regular superset of  $S$ . In general, increasing the number of nonterminals will give better and better “fits”, i.e. smaller supersets, but a perfect fit to a nonregular set is (by definition) impossible.

*References* The papers [Reynolds, 1969] and [Jones and Muchnick, 1981; Jones, 1987] contain methods to construct, given a program involving structured values, a regular tree grammar describing it. Essentially similar techniques, although formalized in terms of tables rather than grammars, have been applied to the lambda calculus [Jones, 1981], interprocedural imperative program analysis [Jones and Muchnick, 1982], a language suitable as an intermediate language for ML [Deutsch, 1990], and “set-based” analyses for the Prolog language [Heintze, 1992].

## 4.5 Abstract interpretation of logic programs

Given the framework already developed, we concentrate on the factors that make logic program analysis different from those seen earlier. Further, we concentrate on Prolog, in which a program is a sequence of *clauses*, each of the form “head  $\leftarrow$  body”:

$$h(t_1, \dots, t_m) \leftarrow b_1(t'_1, \dots, t'_n) \wedge \dots \wedge b_k(t''_1, \dots, t''_p)$$

where  $h, b_1, \dots, b_k$  are *predicate names* and the  $t_i$  are *terms* built up from *constructors* and *variables*. Variable names traditionally start with capital letters, e.g.  $X$ . Constructors are as in functional languages (e.g. “.” and “[]”), but run-time values are rather different, as they may contain free or “uninstantiated” variables. Each  $b_i(t_1, \dots)$  is called a *goal*.

An example program, for appending two lists:

```
append(Xs, [], Xs) ← .
append(X:Xs, Ys, X:Zs) ← append(Xs, Ys, Zs).
```

#### 4.5.1 Semantics of Prolog programs

Prolog can be viewed either as a pure logical theory (so a program is a set of “Horn clauses” with certain logical consequences), or as an operationally oriented programming language. When used operationally, programs may contain features with no interpretation in mathematical logic, to improve efficiency, or facilitate communication with other programs. Examples: input-output operations, tests as to whether a variable is currently instantiated, and operations to add new clauses to the program currently running, or to retract existing clauses.

A *ground* term is one containing no variables. The *bottom-up* or logical interpretation of “append” is the smallest 3-ary relation on ground terms which satisfies the implications in the program. It thus contains  $\text{append}(1: [], 2: [], 1: 2: [])$  and  $\text{append}(1: 2: [], 3: 4: [], 1: 2: 3: 4: [])$ , among others.

*Top-down* interpretation is used for Prolog program execution. Computation begins with a query, which is a “body” as described above. The result is a finite or infinite sequence of *answer substitutions*, each of which binds some of the free variables in the query. In a top-down semantics, the basic object of discourse is not a store or an environment, but a substitution that maps variables to new terms — which may in turn contain uninstantiated variables, i.e. be nonground.

The result of running the program above with an initial query  $\text{append}(1: 2: [], 3: 4: [], Ws)$  would be the one-element answer sequence

$[Ws \mapsto 1: 2: 3: 4: []]$

while the result of running with query  $\text{append}(Us, Vs, 1: 2: [])$  would be the sequence of three answers

$[Us \mapsto 1: 2: [], Vs \mapsto []]$   
 $[Us \mapsto 1: [], Vs \mapsto 2: []]$   
 $[Us \mapsto [], Vs \mapsto 1: 2: []]$

and the result of running with query  $\text{append}(1: 2: [], Vs, Ws)$  would be an answer containing an uninstantiated variable:

$[Vs \mapsto Ts, Ws \mapsto 1: 2: Ts]$

### 4.5.2 Special features of logic programs

The possibility of more than one answer substitution is due to the *backtracking* strategy used by Prolog to find all possible ways to satisfy the query. These are found by satisfying the individual goals  $q_i(\dots)$  left to right, *unifying* each with all possible clause left sides in the order they appear in the program. Once unification with a clause head has been done, its body is then satisfied in turn (trivially true if it is empty). This procedure is often described as a depth-first left-to-right search of the “SLD-tree”.

Bindings made when satisfying  $q_i$  are used when satisfying  $q_j$  for  $j > i$ , so in a certain sense the current substitution behaves like an updatable store. A difference is that it is “write-once” in that old bindings may not be changed. However, further changes may be made by instantiating free variables.

### 4.5.3 Types of analyses

All this makes program analysis rather complex, and the spectrum seen in the literature is quite broad. Many but not all analyses concentrate on “pure” Prolog subsets without nonlogical features. Some are bottom-up, and others are top-down.

### 4.5.4 Needs for analysis

A first motivation for analysis is to optimize memory use—Prolog is notorious for using large amounts of memory. One reason is that due to backtracking the information associated with a predicate call cannot be popped when the call has been satisfied, but must be preserved for possible future use, in case backtracking should cause the call to be performed again. Considerable research on “intelligent backtracking” is being done to alleviate this problem, and involves various abstract interpretations of possible program behaviour.

Another motivation is speed. Unification of two terms containing variables is a fundamental operation, and one that is rather slower than assignment in imperative languages, or matching as used in functional languages. One reason is that unification involves *two-way bindings*: variables in either term may be bound to parts of the other term (or transitively even to parts of the same term). Another is the need for the *occur check*: to check that a variable never gets bound to a term containing itself (although sometimes convenient for computing, programs containing such “circular” terms have no natural logical interpretation).

### 4.5.5 Examples of analysis

*Analyses to speed up unification* The following are useful to recognize when special forms of unification can be used such as assignment or one-way matching:



*Mode analysis* determines for a particular goal those of its free variables which will be unbound whenever a goal is called, and which will be bound as a result of the call [Mellish, 1985].

*Groundness analysis* discovers which variables will always be bound to a ground (variable-free) term when a given program point is reached. For example, “append”, when called with a query with ground Xs and Ys, yields a substitution with ground Zs; when called with ground Zs, all answers will have ground Xs and Ys; and when called with only Xs ground, the answers will never be ground.

Groundness analysis is more subtle than it appears due to possible aliasing and shared substructures, since binding one variable to a ground term may affect variables appearing in another part of the program being analysed. Simple groundness and sharing analyses are described in [Jones and Søndergaard, 1987]. A more elegant method using propositional formulas built from  $\wedge$  and  $\Leftrightarrow$  was introduced by Marriott and Søndergaard [?] and compared with other methods in [Cortesi *et al.*, 1991].

#### *Safely avoiding the occur check*

*Circularity analysis* is a related and more subtle problem, the goal being to discover which unifications may safely be performed without doing the time-consuming “occur check”. The first paper on this was by Plaisted [Plaisted, 1984], with a very complex and hard to follow method. A more semantics-based method was presented in [Søndergaard, 1986].

*Other analyses* The “difference list” transformation can speed programs up by nonlinear factors, and can be applied systematically; but it can also change program semantics if used indiscriminately. Analyses to determine when the transformation may be safely applied are described in [Marriott and Søndergaard, 1989].

Other uses include binding time analysis for offline partial evaluation, and deciding when certain optimizing transformations can be applied. One example is deforestation [Wadler, 1987].

### 4.5.6 Methods of analysis

Analysis methods can roughly be divided into the pragmatically oriented, including [Bruynooghe, 1991; Mellish, 1985], and [Nilsson, 1991]; and the semantically oriented, including [Cortesi *et al.*, 1991; Debray, 1986; Jones and Søndergaard, 1987], and [Marriott *et al.*, 1993].

A natural analogue to the accumulating semantics seen earlier was used in [Jones and Søndergaard, 1987] and a number of later papers, and presumes given a sequence of clauses and a single query. It is a “sticky” semantics in which the program points are the positions just before each clause goal or the query, and at the end of each clause and the query. With each such point is accumulated the set of all substitutions that can obtain there during computations on the given query (so those for the query end



describe the answer substitutions).

*Approximation of substitutions and unification* These are nontrivial problems for several reasons. One is renaming: each clause is (implicitly) universally quantified over all variables appearing in it, so unification of a predicate call with a clause head requires renaming to avoid name clashes, as the same variable may appear in many “incarnations” during a single program execution. Many approximations merge information about all incarnations into a single abstraction, but this is not safe for all analyses.

Aliasing is also a problem, since variables may be bound to one another so binding one will change the bindings of all that are aliased with it. Terms containing free variables have the same problem: binding one variable will change the values of all terms containing it.

Finally, unification binds variables to structured terms, so approximations that do not disregard structure entirely have some work to do to obtain finite descriptions. A recent example is [Heintze, 1992].

[Marriott *et al.*, 1993] is interesting in two respects: it uses the meta-language approach described in this work, in section 3; and it uses sets of constraints instead of substitutions, reducing some of the technical problems just mentioned.

## 5 Glossary

**abstraction function:** Usually a function from values or sets of values to abstract values such as EVEN, ODD. See *adjoined pair*.

**accumulating semantics:** A semantics that models the set of values that a standard semantics (or instrumented semantics) may produce. The functionality of commands might be  $\mathcal{P}(\text{Sto}) \rightarrow \mathcal{P}(\text{Sto})$ , and the program description might be  $\text{Pla} \rightarrow \mathcal{P}(\text{Sto})$ , where Pla is the domain of program points (or places).

**adjoined pair:** A pair of functions  $(\alpha: D \rightarrow E, \gamma: E \rightarrow D)$  that satisfies  $\alpha(d) \sqsubseteq e$  if and only if  $d \sqsubseteq \gamma(e)$ . The first component is often called an *abstraction function* (or a *lower adjoint*) and the second component is called a *concretization function* (or an *upper adjoint*).

**backwards:** Used for an analysis where the program is analysed in the opposite direction of the flow of control, an example being liveness analysis (see *live variable*). In section 3 this is formalized by interpreting  $\Rightarrow$  as  $\leftarrow$  where  $D \leftarrow E$  means  $E \rightarrow D$ .

**collecting semantics:** Has been used to mean *sticky semantics* as well as *lifted* (or *accumulating semantics*), so some confusion as to its exact meaning has arisen in the literature.

**concretization function:** Usually a function from abstract values such as EVEN, ODD to sets of concrete values. See *adjoined pair*.

**context:** A description of how a computed value will be used in the remainder of the computation. Used in backwards analysis of functional programs.

**core semantics:** See factored semantics.

**correctness:** Given a relation of theoretical or implementational importance, correctness amounts to showing that the properties obtained by abstract interpretation always have this relation to the standard semantics.

**duality principle:** The principle of lattice theory saying that by changing the partial order from  $\sqsubseteq$  to  $\sqsupseteq$  one should also change least fixed points to greatest fixed points and least upper bounds to greatest lower bounds and that then dually related information results.

**factored semantics:** The division of a denotational semantics into two parts: a core, assigning terms to every language construct, but with some details left unspecified; and an interpretation, giving the meanings of the omitted parts. Often used to compare the correctness of one abstract interpretation with respect to another abstract interpretation.

**first-order:** Used for an analysis where the properties directly describe actual values. An example is *detection of signs* where the property ‘+’ describes the values 1, 2, 3, etc.

**forwards:** Used for an analysis where the program is analysed in the same direction as the flow of control. In section 3 this is formalized by interpreting  $\rightarrow$  as  $\rightarrow$ .

**independent attribute method:** Used for an analysis where the components of a tuple are described individually, ignoring relationships among components. In section 3 this is formalized by interpreting  $\times$  as  $\times$ .

**induced property transformer:** An analysis that is obtained from a standard semantics or another analysis in a certain way that is guaranteed to produce an optimal analysis over a given selection of properties.

**instrumented semantics:** A version of the standard semantics where more operational detail is included. In general an instrumented semantics constrains the implementation of a language as defined by its standard semantics.

**interpretation:** See factored semantics.

**lax functor:** A modification of the categorical notion of functor in that certain equalities are replaced by inequalities.

**lifted semantics:** Another term for the *accumulating semantics*.

**live variable:** A variable whose value may be used later in the current computation.

- logical relation:** A relation constructed by induction on a type structure in a certain “natural” way.
- minimal function graph:** An interpretation that associates to each user-defined function in a program a subset of  $S^* \times S$  that indicates those argument/result pairs which are actually involved in computing outputs for a given input to a program.
- relational method:** Used for an analysis where the interrelations among components of a tuple are described, e.g.  $X + Y < 110$ . In section 3 this is formalized by interpreting  $\times$  as  $\otimes$  (tensor product).
- representation transformation:** A function that maps values or properties to properties, e.g. an abstraction function.
- safety:** Essentially the same as correctness, but emphasizing that the results of an analysis may be used for program transformation without changing semantics. Often used when the correctness of one abstract interpretation is established with respect to another abstract interpretation.
- second-order:** Used for an analysis where the properties do not directly describe actual values but rather some aspects of their use. An example is *liveness* where the property *live* does not describe any value but rather that the value might be used in the future computations.
- standard semantics:** A semantics where as few implementation considerations as possible are incorporated. The functionality of commands might be  $\text{Sto} \rightarrow \text{Sto}$ .
- static semantics:** Has been used to mean *sticky lifted semantics* but this usage conflicts with the distinction between static and dynamic semantics.
- sticky semantics:** Used for a semantics which binds program points to various information (“sticky” in the sense of flypaper). In a sticky semantics a command might be of functionality  $\text{Sto} \rightarrow \text{Pla} \rightarrow \mathcal{P}(\text{Sto})$ , where Pla is the domain of program points (or places).
- strict function:**  $f : V_1 \times \dots \times V_n \rightarrow W$  is strict in its  $i$ th argument if  $f(v_1, \dots, v_{i-1}, \perp, v_{i+1}, \dots, v_n) = \perp$  for all  $v_i \in V_i$ .
- tensor product:** An operation  $\otimes$  on algebraic lattices that may be used to formalize the notion of *relational method*. When formulated in the categorical framework, as is natural when recursive types are to be considered, the concept of *lax functor* is necessary.

## References

- [Abramsky, 1986] S. Abramsky. Strictness analysis and polymorphic invariance. In H. Ganzinger and N. D. Jones, editors, *Programs as Data Objects*, volume 217 of *LNCS*, pages 1–23. Springer-Verlag, Oct. 1986.

- [Abramsky, 1987] S. Abramsky. *Abstract Interpretation of Declarative Languages*. S. Abramsky and C. Hankin, eds., Ellis Horwood, 1987.
- [Abramsky, 1990] S. Abramsky. Abstract interpretation, logical relations and Kan extensions. *Journal of Logic and Computation*, 1(1):5–40, 1990.
- [Aho et al., 1986] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.
- [Ammann, 1994] J. Ammann. Elemente der Projektions-analyse für funktionen höherer Ordnung. Master's thesis, University of Kiel, 1994.
- [Bandelt, 1980] H.-J. Bandelt. The tensor product of continuous lattices. *Mathematische Zeitschrift*, 172:89–96, 1980.
- [Barendregt, 1989] H. P. Barendregt. Term graph rewriting. In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, editors, *PARLE—Parallel Architectures and Languages Europe*, volume 259 of *LNCS*, Springer-Verlag, 1989.
- [Bloss and Hudak, 1986] A. Bloss and P. Hudak. Variations on Strictness Analysis. In *LISP'86, Cambridge, Mass*, pages 132–142, Aug. 1986.
- [Bondorf, 1990] A. Bondorf. Self-Applicable Partial Evaluation. Ph.D. thesis 90/17, DIKU, Univ. of Copenhagen, Denmark, 1990.
- [Bondorf, 1991] A. Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17: 3–34, 1991.
- [Bruynooghe, 1991] M. Bruynooghe. A practical framework for the abstract interpretation of logic programs. *Journal of Logic Programming*, 10:91–124, 1991.
- Tech. Rep. CW-62, Catholic Univ. of Leuven, Belgium, 1987.
- [Burn et al., 1986] G. L. Burn, C. L. Hankin, and S. Abramsky. Strictness analysis for higher-order functions. *Sci. Comput. Prog.*, 7:249–278, 1986.
- [Burstall and Darlington, 1977] R. M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *J. ACM*, 24(1):44–67, Jan. 1977.
- [Cortesi et al., 1991] A. Cortesi, G. File, and W. Winsborough. Prop revisited: Propositional formulas as abstract domain for groundness analysis. In *Proceedings 6th Annual Symposium on Logic in Computer Science*, pages 322–327, 1991.
- [Cousot and Cousot, 1977a] P. Cousot and R. Cousot. Automatic synthesis of optimal invariant assertions: mathematical foundation. In *Symposium on Artificial Intelligence and Programming Languages*, volume 12(8) of *ACM SIGPLAN Not.*, pages 1–12, Aug. 1977.
- [Cousot and Cousot, 1977b] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *4th POPL, Los Angeles, CA*, pages 238–252, Jan. 1977.



- [Cousot and Cousot, 1977c] P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In *IFIP Working Conference on Programming Concepts*, pages 237–277. North-Holland, 1977.
- [Cousot and Cousot, 1979] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *6th POPL, San Antonio, Texas*, pages 269–282, Jan. 1979.
- [Cousot and Halbwachs, 1978] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th POPL, Tucson, AR*, Jan. 1978.
- [Curien, 1986] P.-L. Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming*. Wiley, 1986.
- [Debray, 1986] S. K. Debray. Dataflow analysis of logic programs. Tech. Rep., Stony Brook, New York, 1986.
- [Deutsch, 1990] A. Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *17th POPL, San Francisco, California*, pages 157–168. ACM Press, Jan. 1990.
- [Dijkstra, 1976] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
- [Donzeau-Gouge, 1981] V. Donzeau-Gouge. Denotational definitions of properties of program computations. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 11, pages 343–379. Prentice-Hall, 1981.
- [Dybjer, 1987] P. Dybjer. Inverse Image Analysis. In *ICALP'87, Karlsruhe, Germany*, volume 267 of *LNCS*, pages 21–30. Springer-Verlag, 1987.
- [Fosdick and Osterweil, 1976] L. D. Fosdick and L. J. Osterweil. Data Flow Analysis in Software Reliability. *Comput. Surv.*, 8(4):305–330, Sept. 1976.
- [Foster, 1987] M. Foster. Software validation using abstract interpretation. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 2, pages 32–44. Ellis-Horwood, 1987.
- [Gordon, 1979] M. J. C. Gordon. *The Denotational Description of Programming Languages: An Introduction*. Springer-Verlag, 1979.
- [Granger, 1991] P. Granger. Static analysis of linear congruences among variables of a program. In *Proceedings of the Fourth International Joint Conference on the Theory and Practice of Software Development (TAPSOFT '91)*, pages 169–192. Volume 493 of *LNCS*, Springer-Verlag, 1991.
- [Hecht, 1977] M. Hecht. *Flow analysis of computer programs*. North-Holland, 1977.
- [Heintze, 1992] N. Heintze. Set Based Program Analysis. PhD thesis, Carnegie-Mellon University, 1992.



- [Hughes, 1986] J. Hughes. Strictness detection in non-flat domains. In H. Ganzinger and N. D. Jones, editors, *Programs as Data Objects*, volume 217 of *LNCS*, pages 112–135. Springer-Verlag, Oct. 1986.
- [Hughes, 1987] J. Hughes. Analysing strictness by abstract interpretation of continuations. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 4, pages 63–102. Ellis-Horwood, 1987.
- [Hughes, 1988] J. Hughes. Backward Analysis of Functional Programs. In D. Bjørner *et al.*, editors, *Proceedings of the Workshop on Partial Evaluation and Mixed Computation*, pages 187–208. North-Holland, 1988.
- [Hughes and Launchbury, 1992] R. J. M. Hughes and J. Launchbury. Projections for polymorphic strictness analysis. *Mathematical Structures in Computer Science*, 2:301–326, Cambridge University Press, 1992.
- [Jensen, 1991] T. P. Jensen. Strictness analysis in logical form. In *Proceedings of the 1991 Conference on Functional Programming Languages and Computer Architecture*, 1991.
- [Jensen and Mogensen, 1990] T. Jensen and T. Mogensen. A backwards analysis for compile time garbage collection. In *ESOP'90, Copenhagen, Denmark*, volume 432 of *LNCS*, pages 227–239. Springer-Verlag, 1990.
- [Jones, 1981] N. D. Jones. Flow analysis of lambda expressions. In *ICALP'81*, volume 115 of *LNCS*, pages 114–128. Springer-Verlag, 1981.
- [Jones, 1987] N. D. Jones. Flow analysis of lazy higher order functional programs. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 5, pages 103–122. Ellis-Horwood, 1987.
- [Jones and Muchnick, 1981] N. D. Jones and S. S. Muchnick. Flow analysis and optimization of Lisp-like structures. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 4, pages 102–131. Prentice-Hall, 1981.
- [Jones and Muchnick, 1982] N. D. Jones and S. S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *9th POPL, Albuquerque, NM*, pages 66–74, Jan. 1982.
- [Jones and Mycroft, 1986] N. D. Jones and A. Mycroft. Data flow analysis of applicative programs using minimal function graphs. In *13th POPL, St. Petersburg, Florida*, pages 296–306, Jan. 1986.
- [Jones and Rosendahl, 1994] N. D. Jones and M. Rosendahl. Higher-order minimal function graphs. In *Algebraic and Logic Programming, ALP'94*, Madrid, Spain, pages 242–252. Volume 850 of *LNCS*, Springer-Verlag, 1994.
- [Jones and Søndergaard, 1987] N. D. Jones and H. Søndergaard. A semantics-based framework for the abstract interpretation of Prolog. In

- S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 6, pages 123–142. Ellis-Horwood, 1987.
- [Jones *et al.*, 1989] N. D. Jones, P. Sestoft, and H. Søndergaard. Mix: a self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.
- [Jones *et al.*, 1993] N. D. Jones, C. Gomard and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, 1993.
- [Kahn, 1987] G. Kahn. Natural semantics. In F. J. Brandenburg, G. Vidal-Naquet and M. Wirsing, editors. *STACS 87. 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany*, pages 22–39. Volume 247 of *LNCS*, Springer-Verlag, 1987.
- [Kam and Ullman, 1977] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977.
- [Kennedy, 1981] K. Kennedy. A survey of data flow analysis techniques. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 1, pages 5–54. Prentice-Hall, 1981.
- [MacLane, 1971] S. MacLane. *Categories for the Working Mathematician*. Springer-Verlag, 1971.
- [Marriott and Søndergaard, 1987] K. Marriott and H. Søndergaard. Notes for a tutorial on abstract interpretation of Prolog programs. In *North American Conference on Logic Programming*, 1987.
- [Marriott and Søndergaard, 1989] K. Marriott and H. Søndergaard. Semantics-based dataflow analysis of logic programs. In *IFIP'89*. North-Holland, 1989.
- [Marriott *et al.*, 1993] K. Marriott, H. Søndergaard, and N. D. Jones. Denotational abstract interpretation of Prolog programs. In *Proceedings of International Computer Science Conference*, pages 206–213. IEEE Computer Society, 1993.
- [Masdupuy, 1991] F. Masdupuy. Using abstract interpretation to detect array data dependencies. In *Proceedings of the International Symposium on Supercomputing, Fukuoka*, pages 19–27, 1991.
- [Mellish, 1985] C. S. Mellish. Some global optimisations for a Prolog compiler. *Journal of Logic Programming*, 2(1):43–66, 1985.
- [Mercouroff, 1991] N. Mercouroff. An algorithm for analysing communicating processes. In *The Mathematical Foundations of Programming Semantics Conference*, Volume 598 of *LNCS*, Springer-Verlag, 1991.
- [Muchnick and Jones, 1981] S. S. Muchnick and N. D. Jones, editors. *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.
- [Mycroft, 1981] A. Mycroft. Abstract Interpretation and Optimising Transformations for Applicative Programs. Ph.D. thesis, Univ. of Edinburgh, Dec. 1981.

- [Mycroft, 1987] A. Mycroft. A study on abstract interpretation and “validating microcode algebraically”. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 9, pages 199–218. Ellis-Horwood, 1987.
- [Mycroft and Jones, 1986] A. Mycroft and N. D. Jones. A relational framework for abstract interpretation. In H. Ganzinger and N. D. Jones, editors, *Programs as Data Objects*, volume 217 of *LNCS*, pages 156–171. Springer-Verlag, Oct. 1986.
- [Mycroft and Nielson, 1983] A. Mycroft and F. Nielson. Strong abstract interpretation using power domains. In *ICALP’83, Barcelona, Spain*, volume 154 of *LNCS*, pages 536–547. Springer-Verlag, July 1983.
- [Naur, 1965] P. Naur. Checking of operand types in algol compilers. *BIT*, 5:151–163, 1965.
- [Nielson, 1982] F. Nielson. A Denotational Framework for Data Flow Analysis. *Acta Informatica*, 18:265–287, 1982.
- [Nielson, 1983] F. Nielson. Towards viewing nondeterminism as abstract interpretation. *Foundations of Software Technology & Theor. Comp. Sci.*, 3, 1983.
- [Nielson, 1984] F. Nielson. Abstract Interpretation using Domain Theory. Ph.D. Thesis, Univ. of Edinburgh, Oct. 1984.
- [Nielson, 1985a] F. Nielson. Tensor products generalize the relational data flow analysis method. In *Proc 4th Hungarian Comp Sci Conf*, pages 211–225, 1985.
- [Nielson, 1985b] F. Nielson. Program transformation in a denotational setting. *ACM TOPLAS*, 7(3):359–379, July 1985.
- [Nielson, 1986a] F. Nielson. A Bibliography on abstract interpretation. *ACM SIGPLAN Not.*, 21(5):31–38, 1986.
- [Nielson, 1986b] F. Nielson. Expected forms of data flow analysis. In H. Ganzinger and N. D. Jones, editors, *Programs as Data Objects*, volume 217 of *LNCS*, pages 172–191. Springer-Verlag, Oct. 1986.
- [Nielson, 1986c] F. Nielson. Abstract interpretation of denotational definitions. *Proc STACS 1986*, volume 210 of *LNCS*, pages 1–20. Springer-Verlag, 1986.
- [Nielson, 1987] F. Nielson. Towards a denotational theory of abstract interpretation. In *Abstract Interpretation of Declarative Languages*, editors S. Abramsky and C. Hankin. Ellis-Horwood, 1987.
- [Nielson, 1988] F. Nielson. Strictness analysis and denotational abstract interpretation. *Information and Computation*, 76:29–92, 1988.
- [Nielson, 1989] F. Nielson. Two-level semantics and abstract interpretation. *Theoretical Computer Science*, 69:117–242, 1989.
- [Nielson and Nielson, 1988a] H. Nielson and F. Nielson. Automatic binding time analysis for typed lambda calculus. *Science of Computer*

- Progaminng*, 10:139–176, 1988. Also see *Proc. 15th ACM Symposium on Principles of Programming Languages*, 1988.
- [Nielson and Nielson, 1988b] F. Nielson and H. R. Nielson. The TML-approach to compiler-compilers. Tech. Rep. ID-TR-193988-47, The Technical Univ. of Denmark, 1988.
- [Nielson and Neilson, 1988c] H. R. Nielson and F. Nielson. 2-level  $\lambda$ -lifting. In *Proc ESOP 1988*, volume 300 of *LNCS*, pages 328–393. Springer-Verlag, 1988.
- [Nielson and Nielson, 1992a] H. R. Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction to Computer Science*. Wiley, 1992.
- [Nielson and Nielson, 1992b] H. R. Nielson and F. Nielson. *Two-level Functional Languages*. Vol 34 of Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1992.
- [Nilsson, 1991] U. Nilsson. Abstract interpretation: a kind of magic. In *Programming Language Implementation and Logic Programming*, edited by J. Maluszynski and M. Wirsing. Volume 528 of *LNCS*, pages 299–309, Springer-Verlag, 1991.
- [Paulson, 1984] L. Paulson. Compiler Generation from denotational semantics. In *Methods and Tools for Compiler Construction*, B. Lorho, ed., pages 219–250, Cambridge University Press, 1984.
- [Plaisted, 1984] D. Plaisted. The Occur-check problem in Prolog. In *Proc. 1984 Symposium on Logic Programming*, pages 272–280, 1984.
- [Plotkin, 1980] G. Plotkin. Lambda definability in the full type hierarchy. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Academic Press, 1980.
- [Plotkin, 1981] G. D. Plotkin. A Structural Approach to Operational Semantics. Tech. Rep. FN-19, DAIMI, Univ. of Aarhus, Denmark, Sept. 1981.
- [Reynolds, 1969] J. C. Reynolds. Automatic computation of data set definitions. In *IFIP'68*, pages 456–461, 1969.
- [Reynolds, 1974] J. C. Reynolds. On the relation between direct and continuation semantics. In *ICALP'74*, volume 14 of *LNCS*, pages 141–156. Springer-Verlag, 1974.
- [Romanenko, 1990] S. A. Romanenko. Arity Raiser and Its Use in Program Specialization. In *ESOP'90, Copenhagen, Denmark*, volume 432 of *LNCS*, pages 341–360. Springer-Verlag, 1990.
- [Rosendahl, 1989] M. Rosendahl. Automatic complexity analysis. In *FPCA '89, London, England*, pages 144–156. ACM Press, Sept. 1989.
- [Schmidt, 1986] D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Newton, MA, 1986.



- [Sestoft, 1988] P. Sestoft. Replacing Function Parameters by Global Variables. M.Sc. thesis 88-7-2, DIKU, Univ. of Copenhagen, Denmark, Oct. 1988.
- [Sestoft and Argo, 1989] P. Sestoft and G. Argo. Detecting unshared expressions in the improved three instruction machine. Tech. Rep., Glasgow Univ., 1989.
- [Sharir and Pnueli, 1981] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–233. Prentice-Hall, 1981.
- [Shivers, 1988] O. Shivers. Control flow analysis in Scheme. In *SIGPLAN '88 Conference on PLDI, Atlanta, Georgia*, volume 23(7) of *ACM SIGPLAN Not.*, pages 164–174, July 1988.
- [Sintzoff, 1972] M. Sintzoff. Calculating properties of programs by valuations on specific models. In *ACM Conference on Proving Assertions About Programs*, volume 7(1) of *ACM SIGPLAN Not.*, pages 203–207, Jan. 1972.
- [Søndergaard, 1986] H. Søndergaard. An application of abstract interpretation of logic programs: occur-check reduction. In *ESOP'86, Saarbrücken, Germany*, volume 213 of *LNCS*, pages 327–338. Springer-Verlag, 1986.
- [Steffen, 1987] B. j Steffen. Optimal Run Time Optimization — Proved by a New Look at Abstract Interpretation. In *TAPSOFT'87*, volume 249 of *LNCS*, pages 52–68. Springer-Verlag, 1987.
- [Stoy, 1977] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.
- [Stransky, 1992] J. Stransky. A lattice for abstract interpretation of dynamic (lisp-like) structures. *Information and Computation*, 101:70–102, 1992.
- [Wadler, 1987] P. Wadler. Strictness analysis on non-flat domains (by abstract interpretation). In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 12, pages 266–275. Ellis-Horwood, 1987.
- [Wadler, 1988] P. Wadler. Deforestation: Transforming programs to eliminate trees. In H. Ganzinger, editor *ESOP '88. 2nd European Symposium on Programming, Nancy, France, March 1988*. Volume 300 of *LNCS*. Springer-Verlag, 1988.
- [Wadler and Hughes, 1987] P. Wadler and R. J. M. Hughes. Projections for strictness analysis. In G. Kahn, editor, *FPCA'87, Portland, Oregon*, volume 274 of *LNCS*, pages 385–407. Springer-Verlag, Sept. 1987.
- [Wegbreit, 1975] B. Wegbreit. Property extraction in well-founded property sets. *IEEE Trans. Software Engin.*, 1:270–285, 1975.



# Index

- Abs 541, 548, 611
  - properties of, desirable 548
- abstraction 150, 196, 225–5, 541, 617–18
  - expansive approach to 319–24
  - full 273, 309, 313
    - order-extensional continuous 319–24
  - function 542, 548, 611, 627
    - nonexistence of 623
    - properties of, desirable 549
  - non-full 319
  - restrictive approach to 319, 324
  - of set of all runs 544
  - of set of stores 544
  - see also* Abs; interpretation, abstract
- abstract value set, *see* Abs
- accumulating semantics, *see under* semantics
- ACP 150, 151, 244–6, 260
  - completeness theorem for 251
  - decidability in 256
  - expansion theorem for 249
  - expressivity in 257–9
  - extension of 251–5
  - semantics of 249–51
    - of ACP 255
- action functions 197, 200–1, 240
- action, atomic 154
  - communication 245
  - potential 240
  - read 245
  - realized 240
- action function 197, 201
- adequacy 273, 309–13
  - Adequacy Theorem 421
    - for Single Sorted Case 422–7
  - weak 309
- Adian–Rabin Theorem 418–19
- adjoints
  - lower 600, 627
  - upper 600, 627
- adjunctions, fibrewise 137–8
- agreement 210
- Alexandrov condition 453
- algebras 358
  - computing in 359–61
    - see also* computations
  - finiteness conditions in 366, 382
  - word problem for 387
- algebras, computable 358–9, 372–3, 379–80, 402
  - $\alpha$ -computable 365
  - constructions of 398–402
  - history of theory of 368–71
  - see also* Completeness Theorems
- algebras, cosemicomputable 366, 379–80
- algebras, countable 361, 386–7, 402
  - definitions of 364–6
  - examples of 361–3, 381–2
- algebras, effective 358
- algebras, finitely generated 393–5
- algebras, Hausdorff, locally compact 472
  - domain representations of 477–8, 478–9
  - effectivity of 494–5
- algebras, higher order 411
- algebras, initial 407
- algebras, many sorted 359
- algebras, metric 472
  - domain representations of 479
- algebras, numbered, with recursive operations 490
- algebras, pre-initial 407
- algebras, representation 367
- algebras, semicomputable 366, 379–80
  - Basic Completeness Lemma for 414–15
  - Completeness Theorems for 415, 445
- algebras,  $\Sigma$ - 374
  - inverse systems of 456
  - inverse limits of 456–7, 458–60
  - surjective 456, 458
  - minimal 375, 430, 432–4
- algebras, simple 401
- algebras, single sorted 359, 360
- algebras, stack 418, 420
- algebras, term 375
  - computable 385–6
- algebras, topological 358, 359, 361, 368, 460

- domain representations of 455
  - construction of 461–5
- effectively domain representable
  - 490–5, 510–11
  - computable elements of 490
- topological subalgebras of 456 *see also* algebras, Hausdorff, locally compact
- algebras, ultrametric 456, 466–8
- algebras, uncountable
  - examples of 363–4
- algebras, universal 369, 373
- algorithms
  - observable 278, 341–4
  - sequential 277, 278, 338–41
- analyses, induced 595–604
  - definition of 596, 604
  - optimality of 602–4
- answer substitutions 624
- approximation indication principle (AIP)
  - 170, 180
  - restricted (AIP) 178, 179
- approximations 446–51
  - definitions of 447, 448
  - effective 480
  - explicit 446, 451
  - induced
    - to base functions 551, 555
  - induction principle 170
  - restricted 178
  - safe
    - of base functions 550
    - of transitions 552
- arity 153
- assertions 226
  - decidable 468
  - definitely false 468
  - definitely true 468
- assignments 376
- associativity 154
- atoms 588
- attributes, independent 564, 628
- autoconcurrency 58
- autoequivalence 396
- automorphism invariants 396
- automorphisms,  $\alpha$ -computable 384–5
- autostability 397
- axiom 153, 160, 209
  - handshaking 249
  - Kleene's 218
  - standard concurrency 233, 239, 247
  - time factorizing 221
  - Tröger's 218
- back-tracking 225, 625
- bags 256, 257
- basic parallel processes 256
- basic term 155, 182, 185, 221
- basis 576
- BCCSP 150
- bifibrations 136
- bindings 625
- bisimulation 163, 213
  - applicative 304
  - bisimulation equivalence 163, 165–6
  - strong 163, 213
- Bohm trees 293–5
- Boone–Higman Theorem 417
- BPA (Basic Process Algebra) 151
  - with discrete relative time ( $BPA_{dt}$ ) 220–4
  - extensions to 224–6
  - with iteration ( $BPA^*$ ) 217–20
  - projection in 169–81
    - in  $BPA + RN$  193
    - in  $BPA_\delta$  183
    - in  $BPA_{\delta\epsilon}$  188, 189
    - in  $BPA_{\delta\epsilon} + RN$  196
  - recursion in 167–9, 178–81, 228
    - in  $BPA_\delta$  183
    - in  $BPA_{\delta\epsilon}$  188–9
    - in  $BPA_{\delta\epsilon} + RN$  196
    - in  $BPA_{\delta\theta}$  216, 217
    - in  $BPA_\Lambda$  203
    - in  $BPA_\lambda$  200
    - in  $BPA_{dt}$  224
  - semantics of 159–67
    - of  $BPA + PR$  172–3
    - of  $BPA + RN$  192
    - of  $BPA^*$  218–19
    - of  $BPA_\delta$  182, 187
    - of  $BPA_\delta + PR$  183–4
    - of  $BPA_\delta + RN$  194
    - of  $BPA_{\delta\epsilon} + RN$  196
    - of  $BPA_{\delta\theta}$  207–13
    - of  $BPA_\epsilon$  185–6
    - of  $BPA_\Lambda$  202
    - of  $BPA_\lambda$  199–200
    - of  $BPA_{dt}$  222–3
    - of  $BPA_{rec} + PR$  179
  - theory of 153–6
- BPP (Basic Parallel Processes) 151, 256
- Branching Lemma 502
- Buffers 251–2
- Cantor set 493

- call
  - system 237
- carriers 359, 374
  - Cartesian liftings 11–12, 136
- category theory 5, 281, 601
- CCS 28, 122, 123, 150, 260
  - NIL 189–90
- CDSs (concrete data structures) 276–7, 328
  - definition of 329
  - deterministic, *see* DCDSs
  - representation theorems for 279, 332
- Ceitin's Theorem 492–3, 510–11
- cells 328, 329, 344–5
  - accessible 330
  - filled 330
  - stale 330
  - well-founded 330
- chains 577
- child process 237
- choice
  - strong 221
  - weak 221
- Church–Rosser property 289
- Church–Turing thesis 326, 360
- circularity analysis 626
- clauses 623–4
- clocked system 230
- closed term 153
- closure 618, 619
  - closure analysis 618–19
  - compatible 288
- cocartesian liftings 12, 136
- coherence 279, 347
  - coherence property 279
  - linear 347
- collecting semantics 609, 614, 627
- combinators 572–5
  - expected forms for 605–9
- communication
  - asymmetric 252
  - asynchronous 240–1
  - binary 245
  - get 253
  - put 252
  - read/send 245
  - synchronous 244
  - ternary 245
- communication action 245
- communication function 244
- communication merge 244–5, 249–50, 255
- commutativity 154
- comp 92–4
- complementation 84, 85
- complete 172
- complete(ness) 166
- completely guarded recursive specification 168
- completely guarded term 168
- Completeness Theorems 177–8, 244, 251, 255
  - First 410–11, 444–5
    - proof of 434–44
  - Second 411–12
    - for Semicomputable Algebras 445
- complete partial order, *see* cpo
- completions
  - ideal 368, 467
  - ultrametric 466
- composition 604–5
  - alternative 154
  - form of, expected 605
  - functional 605
  - parallel 229
    - interleaving 230
    - synchronous 230
  - prefix sequential 256
  - sequential 153, 154
- compositionality 562
- computability 309, 406–7
  - argument 309–12
- computations
  - approximate 368
  - exact 3678
- Conc 549
- conclusion 160, 209
- concretization function 542, 549, 611, 627
- concurrency 63, 228
  - concrete 229
  - standard 233, 239, 247–8
    - axioms for 233–4, 248
- concurrency theory 150
  - algebraic 150
- conditional
  - constructs 226
  - term rewriting system 206
- conditional upper semilattice, *see* cusp
- conflict relation 42
- confluent 172
- congruence 164, 401–2, 457
  - $\alpha$ -semidecidable 401
  - null 401
  - observational 187

- separating 457–8, 492
  - upward consistent 460
  - unit 401
- conn 94
- conservativity 173–8
  - equational 175, 177
  - operational 174–5
    - up to  $\psi$  equivalence 175, 176
- constant propagation 547, 617
- constants 359
- constant symbol 153
- constructors 625
- contact 64
- Context Lemma 302–4
- contexts 286, 619, 620–1, 627
  - applicative 302
  - context substitution 286–7
  - context transformers 621–2
  - evaluation 299, 300
    - raw 299–300
- continuations, current 621
- continuity theorem, syntactic 295
- cooperation 604
- coreflection 34, 54–8, 73–4
- core semantics 561, 563, 629
- correctness 569, 588–95, 603–4, 628
  - of basic expressions 589–90
  - of closed expressions 590
  - of combinators 589–90
- correspondence, correctness 589, 603, 604
- counters 227
- cpos (complete partial orders) 313, 452, 563, 576
  - algebraic 313, 453
    - $\omega$ - 313
    - prime 336
    - strongly 314
  - consistently complete 313, 576
  - projections of, inverse system of 313–14
  - with stable functions 335
- CSP 150
- cul 368, 451
  - of compact neighbourhood systems 4734, 4945
  - ideal completion of 473, 494
  - ideal completion of 452, 453
- data-flow analyses
  - examples of 536–8
- data structure 260
- data types 403
  - abstract 405
  - as algebras 403–5
  - computable 406–7, 408
  - concrete 405
- DCDSs 279, 330, 333, 340, 343
  - filiform 330, 340, 343
- De Simone format 165
- deadlock 181–3, 224, 239
- decidability 226, 255, 468–9
- deduction graphs 161
  - bisimilar 163
- deduction rules 160, 209
  - conclusions of 160
  - hypotheses of 160
  - pure 165, 214
  - well-founded 174, 214
- definability 283
- degree 211
  - $\sim$  of a rule 211
  - $\sim$  of a term deduction system 211
- $\delta$ -redex 288
- $\delta$ -reductions 287
- denotations 561
- dense, effectively 505
- dependency
  - analyses 530–1
  - causal 42
- diamond property 289
- dI-domains 46–7, 125, 277, 336–7, 347
- difference list transformation 626
- discrete order 450
- discrete time unit delay 220
- distributed systems
  - models for, *see* models for concurrency
- distributivity
  - full 154
  - left (LD) 154
  - right 154
- domains 368, 575, 576
  - concrete 279, 332
  - constructive 485
  - domain definitions 562
  - domain orders 453
  - domain projections 621
  - domain representability 446
  - effective 480, 485
    - definition of 480–1
    - presentation of 481
    - subdomains of, constructive 485–90
  - product of, Cartesian 455
  - Scott, *see* Scott domains

- Scott-Ershov 446, 447, 453, 480
- $\Sigma$ -substructures of 455
- structured (E-) 454
- suborders on 594
- totality in 468
- dt** 578, 595, 604
- duality principle 628
- effect functions 197, 201, 240
- elements
  - canonical representative 472
  - compact 452–3, 577
  - computable 481
  - effective 481
  - finite 345, 452–3
  - maximal 456, 470
  - recursive 481
  - total 469, 470, 471–3, 475
  - space of 472
- elimination property 171
- elimination theorems 171, 199, 207, 233, 243, 247
- embeddings 102–6, 462
  - of asynchronous transition systems 104–5
  - of event structures 106, 108–9
  - of nets 103–4
  - rigid 103, 106
  - of trace languages 105
- empty process 185, 241
  - conservativity 186–8
- enabling relations 329
- encapsulation 194–5, 197, 241, 244, 252, 255
  - operator 195
- endofunctions 146, 147
- enumeration
  - computable 364–5
  - enumeration functions 434–5
- environments 306
  - undefined 306
- equation
  - recursion 167
  - recursive 167
  - theory of equations 377
- equational specifications 153, 411–14
  - enumerably, recursively 414
  - equational hidden enrichment 412
  - finite 413
  - methods of
    - adequate 409
    - complete 409
    - Completeness Theorems for 409–11
    - sound 409
    - recursive 414
    - sum of two  $\sim s$  177
    - and term rewriting systems 429–3
- equationally conservative extension 177
- equivalence
  - bisimulation 163, 213
- equivalence relations
  - decidable 365
- equivalences 176
- error propagation 343
- error values (errors) 278, 342
- evaluation, partial 531, 620
- events 37, 41, 106
  - idling 62
  - independence of 71
  - labelled partial orders of 124
  - minimum representatives of 52
  - see also* event structures
- event structures 41–61, 125–6
  - associated with trace languages 47–61, 126
  - coreflection between categories 54–8, 120–1
  - definition of 52
  - computation state of 43
  - concurrency relation of 53–4
  - configurations of 43
  - coproduct of 45
  - definition of 42
  - domains of configurations 45–7
    - bounded complete 46
    - coherent 46
    - finitary 46
    - prime algebraic 46
  - embeddings of 106, 108–9
  - flow 125
  - labelled 58, 106–9, 118, 120–2
    - coproducts of 107–8
    - coreflections between categories of 118
    - products of 107
    - recursion on 108–9
    - restriction of 108
  - morphisms of, *see under* morphisms
  - product of 45, 557–8
- expansion 234, 239, 248
- expansions 375
- expansion theorems 234, 249
- expressions 570
  - closed 575
  - interpretation of 579–82
  - raw 285



- syntax of 572
  - well-formed 573–4, 578–9
- expressivity 226, 256
- extended state operator 201
- extensionality 326
- factored semantics 563–5, 628
- fibrations 136
- fibre 136
  - coproducts in 17, 18
  - products in 15
- field extensions, canonical 369
- FIFO queue 258
- fixpoints 556
- fork 237
- format
  - De Simone 165
  - GSOS 208
  - ntyft/ntyxt 212
  - panth 212
  - path 165
  - tyft/tyxt 164
- forms, expected 604–9
- formulas 160
  - $a \sim$  holds in ... 209
  - negative 209
  - positive 109
- $f_0$ -spaces, constructive 485
- free merge 229
- full distributivity 154
- function
  - on abstract values 616–17
  - action 197, 201
  - additive 576
    - binary 576, 587
  - adjoined 597
  - communication 244
  - compact preserving 576
  - computable 496, 497, 498, 500, 510
    - effectiveness of 499–501
    - index for 497
  - computable  $(\alpha, \beta)$ - 492
  - conditionally multiplicative (CM) 335
  - continuous 453, 464, 471–2, 492, 504, 511, 576
    - continuous effective 484, 489
    - effectively 504, 509–10
    - representation of 478
  - effective,  $(\alpha, \beta)$ - 482, 492, 504
  - effect 197, 201
  - function description table, global 618–19
  - function graphs of, minimal 617–18, 629
  - higher-order 618–19
  - input-output, observable 342
  - inverse, weak 600–1
  - $\lambda$  -congruent 463
  - linear 47
  - monotonic 576
  - process creation 236
  - recursive 360, 372, 496
    - partial 492, 505
    - total 486, 487–8, 503, 505
  - renaming 190
  - semantic 562
  - sequential 280
    - observably 278, 341, 343
  - stable, strongly 347
  - strict 576, 629
- function space, strict 582
- function symbols 153
  - with lexicographical status 158
- functors
  - between categories of models 118–22
  - forgetful 120
  - lax 628, 629
- games, categories of 126–7
- get
  - communication 253
  - mechanism 252
- goals 624, 625
  - Gödel's System **T** 282
- Goncharov's Theorem 396
- grammars 6223
- graphs
  - deduction 161
  - enumeration of 422–3
  - variable dependency 174, 214
- groundness analysis 626
- groups 383
  - finitely presented 415–18
  - Markov property of 418
  - word problem for 416
- GSOS format 208
- guard 168, 226
- guarded
  - command 226
  - occurrence 168
  - recursive specification 168
  - term 168
- handshaking 245–6

- handshaking axiom (HA) 249
- hiding 124
- Hindley–Rosen Lemma 289
- Hoare languages 61
- Hoare traces 32, 123
- hold (a formula  $\sim$ s in ...) 209
- homomorphisms 532
- Homomorphism Theorem for algebras 366
  - Computable Homomorphism Theorem 401
- hypothesis 160, 209
- ideals 474–6
  - principle 452, 475
- idempotency 154
- if-then-else operator 225
- inaction 181
- inclusions 34, 486–7
- independence relation 37
- indeterminates 375
- index, finite 402
- induction
  - structural 155, 172
- information flow 620–1
- instrumented semantics, *see under* semantics
- interleaving 230
- interleaving parallel composition 230
- interpretation 561, 563, 580
  - abstract 528
    - applicability of 568
    - correctness of analyses, *see* correctness
    - Cousot approach to 611–12
    - expected forms of analyses 604–9
    - extensions to 609–10
    - induced analyses, *see* analyses, induced
    - origins of 535–6
    - semihomomorphic 533
    - specification of 575–81
  - accumulating 563
  - domain 577, 580
  - even-odd 564–5
  - frontier 610
  - lattice 577, 580, 605–9
  - program 611
  - standard 563, 565
  - time 615
  - top-down 624
  - of types 577–8
- intersection property, finite 474–5
- Invariance Theorem 395
- invariants 226
- invertibility, weak 600–2
- isomorphism types 405
- it 593, 595, 600
- iteration 217, 236, 242, 253
- Jung–Stoughton criterion 276
- Kahn–Plotkin sequential functions 277–8, 333, 338, 344
- Kan extensions 600, 613
- Kleene’s axiom 218
- Kleene’s binary star operator 217–8, 253
- Kleene’s problem 280
- Kreisel–Lacombe–Shoenfield Theorem 505, 506
  - generalisation of 505–9
- labelled transition system 162
- labelling operation 119, 122
- labels 37, 41, 106
- $\lambda$ -calculus 281, 304
  - two-level 571, 575
  - typed 569, 575
- $\lambda$ -lifting 575
- $\lambda$ -redex 288
- $\lambda$ -reductions 287
- languages 32–4
  - category of 58
    - reflection from, to category of synchronisation trees 58–61
  - coproduct of 34
  - denotational, definitions of 562
  - fibre product of 34
  - functional 611
    - analysis of 616–21
    - lazy 615
  - imperative 611
  - logic programming 611, 624
  - morphisms of, *see under* morphisms
  - prefixing for 35
  - product in 35
  - recursion in 35
  - theory of, classical 125
  - see also* Hoare languages; trace languages
- lattices 542–3
  - algebraic 576, 577
  - complete 548, 560, 575
  - dual 556
  - nonetherian 543
- left distributivity 154

- left merge 229
- lexicographical path ordering 158
- lexicographical status 158
- limits, inverse 456–7, 464
  - construction of 458–60, 492
  - examples using 493–4
- linear recursion equation 227
- linear recursive specification 227
- listing, complete 599
- liveness 584–5, 591–2, 629
- lock step 230
- look-ahead 196
- lpt** 593, 595, 598, 600, 602, 604
- lt** 578
  
- Macintyre–Neumann Theorem 417
- maps
  - $(\alpha, \beta)$ -computable 383–4
  - linear 345
  - postcondition 62, 63
  - precondition 2, 63
- markings 62, 96
  - reachable 62
- meanings 561
- merge 229
  - communication 244, 249–50, 255
  - free 229, 243–4
    - axioms for 231
  - left 229, 230, 241, 243–4
- metalanguage 538–9, 612
  - pragmatics of 575
  - role of 568
  - syntax of 569–75
  - two-level 572
  - see also* expressions; types
- Milner's expansion theorem 31
- modality 560
- mode analysis 626
- models
  - abstract 274
  - for concurrency 1–7, 122, 128
    - checking of 128
    - equivalences on 128
    - independence 121, 122
    - interleaving 3, 121
    - noninterleaving 3
  - synthetic descriptions of 274–5
- morphisms 5
  - cartesian 11, 136, 137, 138
    - strong 136, 137, 138
  - cocartesian 136
  - counit 81, 82
  - definition of 9
    - folding 98
    - of event structures 44, 54, 58, 108
    - of languages 33
    - partial simulation 123–4
    - on (Petri) nets 65–6, 68, 82, 126
      - relations 96–102
    - of trace languages 39, 54–5, 59
    - of transition systems 72, 110
      - of asynchronous transition systems 74, 78, 82
    - unit 81, 82
    - vertical 136
    - zig-zag 26
- multirelations 96
- multisets 96
  - multiset status 158
- mu-notation 167
- Myhill–Shepherdson Theorem 489, 501–5
  
- narrowing 547
- natural numbers ( $\mathbb{N}$ , or  $\omega$ ) 366, 480, 493
  - completion of 461
  - partial function on 501, 505
  - partial recursive functions on 501, 505
  - total recursive functions on 505
- necessary termination predicate 189
- need analysis 531
- negative formula 209
- nesting operator 225
- nets, *see* Petri nets
- NIL 189–90
- nondeterminism 64, 155, 534–5
- normal form 156
  - approximate 293
  - partial 293
- numberings 377–8, 380
  - canonical 486, 487
  - complete, recursively 488, 489
  - computable 378–9, 387–8, 391
    - standard 385
  - constructive 485, 487–8
  - cosemicomputable 379
  - effective 378, 380
    - recursive equivalence of 389–96
    - reductions between 388–9
  - with recursive operation 490
  - semicomputable 379, 391
- nyttf/ntyxt format 212
  
- object names 197, 201
- observables 271

- observational equivalence 271, 302, 304, 308
  - definition of 272
- occur checks 625, 626
- occurrences 292
  - guarded 168
  - unguarded 168
  - occurrence net 102
- one-place buffer 251
- op 550, 551
- open term 153
- operationally conservative extension 174, 176, 214
- operations 359
  - non-expansive 466
- operator
  - encapsulation 195
  - extended state 201
  - Kleene's binary star 217
  - nesting 225
  - priority 204
  - process creation 237
  - projection 170
  - rank of an  $\sim$ , 199, 232, 246
  - renaming 190
  - restriction 194
  - signal insertion 225
  - signal termination 225
  - simple state 197
  - unless 204
- optimization 531, 535, 536, 612, 613
  - example of 546
- ordering 555–6
  - extensional 278
  - priority 204
  - stable 277, 278, 336
- PA 151, 229–34
  - completeness theorem for 235
  - projection in 236
  - recursion in 235, 236
  - semantics of 234–5
    - of PA + PCR 238
    - of  $PA_\delta$  240
    - of  $PA_{\delta dt}$  243–4
    - of PAreC 236
- panth format 212
  - congruence theorem for 213
- parallel-and 320
- parallel composition 224, 229
  - interleaving 230
  - synchronous 230
- parallel conditional 323
- parallel-or 320, 323
- parent process 237
- parity analysis 538
- partiality, local 328
- path convergence 556
- path format 165
  - congruence theorem for 165
- path ordering, recursive 157, 158
- PCF (Programs for Computable Functions) 270, 283–4, 299–301
  - denotational models of 304–8
    - continuous models 306–7, 327
    - order-extensional continuous 313–18
    - structural constraints 307–8
  - denotational semantics of
    - compositional least fixed-point 307
    - compositional nature of 312
  - extended with errors 344
  - finitary 276
  - full abstraction for 274–6
    - intensional 275–6
  - operational semantics of 300–1
  - overview of 272–4
  - with ‘parallel-or’ (PCFP) 319, 325
  - PCF-sequentiality, *see under* sequentiality
  - Plotkin's definition of 301
  - standard (continuous) model for 272–4
  - value domains for 305
- Petri nets 61–70, 126–7
  - and asynchronous transition systems, *see under* transition systems
  - category of 65–6
  - condition-extensional 90
  - condition of 76–80, 81
    - extent of 76, 77
    - reachale extent of 88
  - constructions on 66–70
    - coproduct 66–8
    - product 69–70
  - definition of 62
  - embeddings of 103–4
  - general 96, 97
  - labelled 121
  - morphisms on, *see under* morphisms
  - safe 64, 68, 86–7, 97
- place 328, 344
- pomset languages 53, 61, 124–5
- port 245

- positive formula 209
- potential action 240
- predicates 557, 576
  - admissible 576
  - domain type 578
  - extensional view of 558
  - impure type 594
  - intensional view of 558
  - lattice type 578
  - level-preserving type 593
  - necessary termination 189
  - predicate names 624
  - predicate transformers
    - backward 559, 561
    - forward 559, 561
  - pure type 593
  - set of 162
  - successful termination 160
  - termination option 186
- predicate symbol 160, 209
- prefix sequential composition 256
- preorders 448
- principle
  - approximation induction 170
  - recursive definition 168
  - recursive specification 168
  - restricted approximation induction 178
  - restricted recursive definition 168
- priority operators 204–17, 241, 252
  - derivation rules for 208
  - ordering 204
  - rewrite rules for 207
- probabilities 226
- Proc** 20, 35, 109, 115–18, 145–8
- process algebra 150, 151
  - basic, *see* BPA
  - concrete 150, 151
  - real-space 225
  - real time 225
  - textbooks on 260
- processes 150, 153
  - child 237
  - concrete concurrent 228
  - concrete sequential 152
  - deadlocked 181
  - empty 185, 241
  - parent 237
  - regular 227
  - subprocesses 227
  - sum of two  $\sim$  ses 154
- process creation 236–9, 242, 253
  - function 236
  - operator 237
- process languages 20
- products
  - cartesian 582, 608
  - smash 582
  - tensor 586–7, 608, 629
- program analysis 528
  - naive 540–1
  - need for approximation in 529–30
- program approximation, safe 623
- program contexts 271–2
- program equivalence 270–1, 308–9
  - see also*, observational equivalence
- program execution
  - nonstandard 528
  - traces of 615
- program languages 283–4
- program preconditions 559
- program postconditions 559
- program run times 615
- programs
  - functional
    - backwards analysis of 560
  - imperative 539
    - backwards analyses of 559–60, 619, 621
  - logic
    - abstract interpretation of 623–7
  - nondeterministic 535
- program termination 546–7, 561, 569, 610
- program transformations 535, 610
- program verification 535, 542, 559
- projection operators 170
- projection theorem 178–9
- Prolog 539, 623–5
- proof 160
- property
  - elimination 171
  - transfer 163, 213
  - transformers, induced 570, 628
- provable 161
  - pure rule 165, 214
- pseudoevaluation 535–6
- pt** 593, 600
- put
  - communication 252
  - mechanism 252
- queues 258–9
  - FIFO 258
- rank 232, 246



- ( $\sim$  of an operator) 199, 232, 246
- rational numbers ( $\mathbb{Q}$ ) 447–8, 495, 497
- reach 161, 227
- reachable 161, 227
- reachability 548
  - definition of 161
- reactive systems 2
- read action 245
- read/send communication 245
- real numbers ( $\mathbb{R}$ ) 445, 447–8
  - computable reals 496
  - computable sequences of 496
  - domain representation of, effective 495–501
- real space 225
- real time 225
- realized action 240
- recursion 282
  - recursion equations 167
  - recursion functions, primitive 436
  - recursion theorems 282, 501
  - recursion theories 367–8
    - admissible 367
    - over algebras 368
    - type 2 367
  - solution of  $a!$  167
- recursive definition principle (RDP) 168, 169
- recursive equations 167
- recursive functions, *see under* functions
- recursive path ordering 157
- recursive specification 167
  - completely guarded 168
  - guarded 168, 178, 227, 228
  - linear 227
  - solution of  $a!$  167
  - unguarded 168, 253
- recursive specification principle (RSP) 168, 169, 181
- redexes 429
- reducibility
  - candidates of 312
  - many-one 388
- reduction maps 388
- reduction rules 156
- reduction systems 283, 427–9
  - algebraic 428
  - Church–Rosser 427
  - complete 427
  - confluent 427
  - $\lambda_y$  ( $\mathbf{A}$ ) 284
    - interpretations of 305
    - operational properties of 288–99
    - reduction rules of 287–8
    - sequentiality of 298–9
    - stability of 299
    - syntax of 284–7
    - normal forms for 427
    - strongly terminating 427
    - weakly terminating 427
- Reduction Theorem, Basic 393–4
- reducts 375, 427
- refinement orders 448
- reflection 34
- regular process 227
- relabelling 12–13, 109, 114
- relational method 629
- relations 557
  - admissible 590
  - logical 312, 566, 612, 629
    - sequential 325
  - on  $n$ -tuples of numbers 622
  - set of  $\sim$   $s$  162
  - transition 160
- relation symbol 160, 209
- renaming 190–7, 217
  - in ACP 252
  - in BPA 190–3
  - in BPA with deadlock 193–4
  - in BPA with empty process 195–6
  - in PA 236
  - in PA with deadlock 241
  - renaming function 190
  - renaming operators 190, 197
- replacement systems, *see* reduction systems
- representations 367
  - faithful 463, 464
  - representation functions 533
  - Representation Lemma 3933
  - Representation Theorem 452
    - see also* algebras, representation
- restricted approximation induction principle 178
- restricted recursive definition principle 168
- restriction 10–12, 114, 194
  - operator 194
- rewriting rule 156
- Rice–Shapiro Theorem 501, 504
- right distributivity 154
- rings, polynomial 400
- rules
  - deduction 160, 209

- degree of  $a \sim$  211
- pure 165, 214
- rewriting 156
- semantic 562
- Safe** 97–101, 121
- safety 530, 547, 549–52, 560, 566–9, 592, 613
  - composition of safety diagrams 554–5
  - definitions of 549–50, 566, 629
  - from denotational viewpoint 566–7
  - local conditions for 566–8
  - safety correspondence 592
  - violations of 531–2
- Scott-continuous function space model 272, 283
- Scott condition 453
- Scott domains 313, 320
  - ordering in 555–6
- Scott topology 453, 501
- semantics 102–18
  - accumulating 538, 542, 627
    - backward 559–60
    - for imperative programs 542–3, 559
    - as an interpretation 565
  - of ACP, *see under* ACP
  - action 575
  - of BPA, *see under* BPA
  - collecting 609, 614, 627
  - core 561, 563; 628
  - denotational 23, 24, 26, 102, 117, 124, 270
    - aim of 271
    - factoring of 563–5, 628
    - limitations of 613–14
    - principles of, basic 561–2
  - function graph, minimal 619
  - instrumented 539, 613–14, 628
    - examples of 614–16
  - lifted 628
  - noninterleaving 127
  - operational 22–6, 102, 109–18, 124, 270, 561, 614
    - aim of 271
    - proofs on 138–48
    - structural (SOS) 124, 270
  - of PA, *see under* PA
  - of PCF, *see under* PCF
  - of Prolog programs 624
  - relationship between 34–6
  - standard 581, 629
    - eager 582
    - lazy 581–2, 591
    - static 629
    - sticky 629
- semigroups
  - finitely generated 416–17
  - word problem for 416, 417
- separation proposition 291–2
- sequentiality 279, 295–9, 325, 333
  - degree of 334
  - features of 295
  - first-order 276, 325
    - definitions of 326–7
  - higher type 275, 280, 325
    - definition of 276
  - PCF<sup>-</sup> 325
    - characterization of 344–8
  - sequential algorithms 334
  - sequentiality index 296, 333
  - sequential prestructures 345
  - sequential structures 279, 345–7
  - topological approach to 344, 348
- send action 245
- sequential composition 154
- Set** 135
- sets
  - $\alpha$ -decidable 383
  - $\alpha$ -semidecidable 502
  - $\gamma$ -semidecidable, weakly 488
  - crisp 321
  - directed 452
  - effectively open 502
    - index of 502
  - open, families of 469
    - separating 469
  - predicates of 162
  - relations of 162
  - states of 162
  - total 469, 470, 471
  - total dense 471
- SFP domains 314
- signal 225
  - insertion operator 225
  - stable 225
  - termination operator 225
- signatures 152, 153, 232, 244, 359, 373–4
  - finite 373
  - non-void (/instantiated/sensible) 373
  - single sorted 373
  - subsignatures 374
  - sum of two  $\sim$  s173

- signs
  - detection of 582–3, 585–8, 591–2
  - rule of 529, 530, 552–4
- simple state operator 197
- size function 139–40
- solution 167
  - unique 168
- soundness 166
- SPCF 278, 341
- specialisation orders 450, 453
- specification
  - equational, 153
  - recursive 167
  - transition system 160
- specifications, equational, *see also* equational specifications
- spectra
  - computable 395, 396
  - effective 394
  - semicomputable 395
- stable signal 225
- stability 277, 299, 334, 335, 338
  - computable 395
  - linear 347
  - semicomputable 395
  - strong 279, 347–8
- stacks 418–21
  - of extended data 419
  - stack algebras 419, 420
  - standard stack polynomials 419
- standard concurrency 233, 239, 247
- standard semantics, *see* semantics, standard
- state
  - set of  $\sim s$  162
  - space 197, 201
  - transition diagram 161
- state operators
  - extended 201–4, 236, 252
  - rewrite rules for 202–3
  - simple 197–201, 236, 252
  - rank of 198–9
  - rewrite rules for 198
- states computation 328
  - sets of 535
- state system, structured 162
- state transition functions 532
- status
  - lexicographical 158
  - multiset 158
- step 154
  - lock 230
- stickiness 611
- sticky semantics 629
- storage reclamation 620
- store descriptions, abstract 543
- store properties 615–16
- stores, sets of 556
- stratifiability 210, 214
- stratifications 210, 212
- strictness 531, 583–4, 616–17, 620
- strong bisimulation 163, 213
- strong choice 221
- strongly normalizing 156
- structural induction 155
- structure
  - data 260
  - sequential 279, 345–7
- structured state system 162
- subalgebras
  - $\alpha$ -computable 398
  - $\alpha$ -decidable 399
  - $\alpha$ -semidecidable 398
- subprocess 227
- subsets 383
- substitution 153
  - closed computable 310
- successful termination 181, 184
- successful termination predicate 160
- sum
  - of two equational specifications 177
  - of two processes 154
  - of two signatures 173
  - of two term deduction systems 174
- symbol
  - constant 153
  - function 153
  - predicate 160, 209
  - relation 160, 209
- synchronous communication 244
- synchronous parallel composition 230
- synchronisation algebras 15–16, 124
  - example of 26–7
- synchronisation trees 6, 28–32, 123
  - category of
    - reflection from category of languages to 58–61
  - definition of 29
  - product of 31
- system
  - clocked 230
  - conditional term rewriting 206
  - labelled transition 162
  - pure 165, 214
  - structured state 162
  - term deduction 160, 209

- term rewriting 156
- system call 237
- tag function 139–40
- tagged terms 23–4, 139
- term deduction systems 159–62, 209
  - agree with a  $\sim$  210
  - degree of 211
  - extensions of
    - with negative premises 209
    - operationally conservative 175, 214–15
  - operationally terminating 219
  - for PA 235
  - pure 165, 214
  - sum of 173–4
  - well-founded 174, 214
- term rewriting systems 156–9, 171–2
  - for ACP 246–7
  - for ACP<sub>dt</sub> 254
  - for BPA + RN 190–1
  - for BPA<sub>seq</sub> 206
  - for BPA<sub>Δ</sub> 202
  - for BPA<sub>λ</sub> 199
  - for BPA<sub>dt</sub> 221
  - Church–Rosser 431
  - complete 171, 410, 429, 444
  - conditional 206
  - equational (equational TRS)
    - 429–31
  - hidden functions in 431
  - left-linear 430
  - non-overlapping 430
  - orthogonal 431
  - for PA 231
  - for PA + PCR 238
  - for PA<sub>seq</sub> 242
  - strongly normalizing 157, 158, 159
- terms 153, 375, 625
  - A -basic 222
  - basic 155, 182, 185, 221
  - closed 153, 155, 166, 286, 375, 408
  - ground 624, 626
  - guarded 168
    - completely 168
  - open 153, 166
  - strongly normalizing (SN) 156
  - term evaluations 376
  - term substitution 286
  - unguarded 168
  - untyped 285
  - weight of 198, 232, 246
- terminating 156
- termination
  - successful 181, 184
  - unsuccessful 181, 184
- termination option predicate 186
- ternary communication 245
- theory 153
- time
  - discrete 220
  - real 225
- time factorizing axiom 221
- time slices 220
- time unit delay, discrete 220–1, 242, 254
- tool support 260
- topology 449
  - quotient 472–3, 476
  - topological spaces
    - open sets of 468
    - $T_0$  449–50
    - $T_1$  451
    - see also* algebras, topological
- totality 455, 468
- trace languages 36–41
  - alphabet of 37, 40–1, 58
  - coherence axiom on 37, 50
  - constructions on 40–1
    - coproduct 40
    - product 40
    - relabelling 40
    - restriction 40
  - embeddings of 105
  - generalisation of, by asynchronous
    - transition systems 73–5
  - Mazurkiewicz 37, 61, 124, 125
  - morphism of, *see under* morphisms
  - order of 39, 41
  - preorder of 47
  - representation theorem for 52–3
- traces 32, 38, 337
  - incompatibility between 49
  - ordering on 38
  - see also* Hoare traces; Maurkiewicz traces
- transfer property 163, 213
- transformations, representation 598, 599
  - 610, 629
- transformers, representation 598, 599, 610
- transitions
  - idle 9, 22, 112–13
  - relation 160
  - system specification 160
  - transition rules 159
- transition systems 4–5, 7–20

- acyclic 8, 139
- asynchronous, *see* transition
  - systems, asynchronous
- category of 10
- construction on 10–22
  - coproduct fibre 113
  - parallel compositions 15–16
  - prefixing 19–20
  - product 13–15, 22, 112
  - relabelling 12–13, 109, 114
  - restriction 10–12, 114, 194
  - sum 17–19
- extensional 7
- generalisations of 121
- with independence 110–12, 121, 127, 145–8
  - operational semantics for 112–18
  - unicity property of 114
- labelled 28–30, 122–3
  - extensions of 123
  - unfolding of 29, 30
- morphism of, *see under* morphisms
- nil 14–15
- reachable 8
- recursion in, guarded 140–5
- transition systems, asynchronous 70–102, 127
  - category of 72, 127
  - conditions of 91–102
    - components of 92–4
    - connected 92–5
    - properties of 92
  - constructions on 72–3, 89–90
    - coproduct 73, 90
    - product 72–3, 89
  - definition of 71
  - embeddings of 104–5
  - extensional 111
  - labelled 110, 121
    - as models for concurrency 101–2
  - and nets
    - adjunction between 76–84, 127
    - coreflection between 84–91
  - and trace languages 73–5
- transversals 393, 408, 428–9
- Tröger's axiom 218
- two-place buffer 251
- tyft/tyxt format 164
- types 284, 570, 609
  - analysis of 576
  - base 570, 609
  - function, *see* functions
  - interpretations of, *see* interpretation
  - notation for 571–3
  - product, *see* products
  - rak of 284
  - recursive 609
  - sum 609
  - two-level 571
  - well-formed 579
- ultrametrics 465
- ultrametric spaces 465, 466
- unification 625, 627
- undecidable 255
- unguarded occurrence 168
- unguarded recursive specification 168
- unguarded term 168
- unique solution 168
- UNIX 237
- unless operator 204
  - derivation rules for 208
  - rewrite rules for 206
- unsuccessful termination 181, 184
- untag 26
- value descriptions 528
  - safe 530, 534
- values
  - abstract 528, 556–8
    - complex 621–3
    - independent attribute analyses 556–7
    - viewed as relations or predicates 557–89
  - context of 620
- variable dependency graph 174, 214
- variables 375, 624
  - live 560, 628
- vending machine 27–8, 64–5
- weak choice 221
- weight of a term 198, 232, 246
- well-founded 174, 214
- well-founded deduction rule 174, 214
- well-founded term deduction system 174, 214
- widening operators 547
- word problem 386
  - for algebras 387
  - for groups and semigroups 416, 417



















3 9001 03413 3275





ISBN 0 10 338730 8